Solidity

SOLIDITY

Steven Walbr...

Solidity is one of several object-oriented, high-level languages used to write Ethereum smart contracts that run on the EVM. Smart contracts are self-executing programs which dictate the behavior of accounts within the Ethereum state.

A contract can contain:

- Functions
- Modifiers
- State Variables
- Events
- Structs
- Enums
- Constructors
- Errors

There also special "Library" contracts and "Interface" contracts to extend or utilize logic outside its own functions.

Solidity contracts are written to a `.sol` file.

STRUCTURE OF A SOLIDITY PROGRAM

Steven Walbr...

SansCoin.sol

```
1  pragma solidity ^0.4.24;
2  contract SansCoin {
3
4    enum UserType {CoinHolder, Admin, Owner}
5
6    struct AccountInfo {
7      address account;
8      string accountName;
9      UserType userType;
10   }
11
12   mapping (address => uint256)
13     public coinBalance;
14   mapping (address => AccountInfo)
15     public registeredAccount;
16
17   address public owner;
18
19   event Transfer(address indexed from,
20     address indexed to, uint256 value);
21
22   modifier onlyOwner {
23     require(msg.sender == owner);
24     _;
25   }
```

version and name

enum definition

struct definition

state variable definitions

event definitions

function modifier definitions

SansCoin.sol

```
27   constructor(uint256 _initialSupply) public {
28     owner = msg.sender;
29     createCoin(owner, _initialSupply);
30   }
31
32   function transfer(address _to, uint _amount)
33     public {
34       require(validateLimit(_amount));
35       emit Transfer(msg.sender, _to, _amount);
36     }
37
38   function validateLimit(uint256 _amount) private
39   returns (bool) {
40     if (_amount < maxLimit)
41       return true;
42     return false;
43   }
44
45   function createCoin (address _recipient,
46     uint256 _createdAmount)
47     onlyOwner public {
48       coinBalance[_recipient] += _createdAmount;
49       emit Transfer(owner, _recipient, _createdAmount);
50     }
```

constructor definitions

function definitions

functions defined with modifiers

you know, makes up, or or what composes and comprises a

GLOBAL NAMESPACE

Steven Walbr...

Variables and functions that are implicitly declared and can always be directly referenced by contract code:

Variable	Function or Property method	Description	Returned Type
msg	data	Body of calldata.	bytes
msg	sender	Address of sender performing the call.	address
msg	gas	The Remaining amount of gas.	uint
msg	value	Amount of Ether sent in the message.	uint
tx	gasprice	The gas price of the transaction.	uint
tx	origin	Address of the originating sender of the chain.	address
block	timestamp	UNIX epoch of a blocks timestamp.	uint
block	number	The block's number.	uint
block	gaslimit	The block's gaslimit.	uint
	selfdestruct	Terminate use of contract on blockchain.	
	require	Boolean used to validate function input.	
	assert	Boolean used to validate state.	

Smart Contract Hacking: Solidity

These can always be referenced. Message has to do with

zoom

5

STATE VARIABLES - REFERENCE TYPES

Steven Walbr...

Variables accessed through their location (reference:). Reference: type variables are located in three areas:
Storage: values permanently persisting on the blockchain
Memory: non-persistent values in temporary memory
Calldata: non-persistent values of externally called function parameters

Reference:TYPE	Description	Example
static arrays	Arrays that are declared with a size	`int32[3] memory myArray;` `myArray[0] = 554;`
dynamic arrays	Arrays that are declared without parameters	`int32[] memory myDynArray;` `myDynArray.push(554);`
string	Initialized with a n length of bytes	`string name = "Steven";`
struct	User defined set of elements with various types	`struct Course { uint CourseId;` `string CourseName;}`
mapping	Key-Value pairs of any type in Storage area.	`mapping(address=>int) public` `myAdd;`

zoom

ACCESS LEVELS

Steven Walbr...

Variable	Description
public	Default for functions. Enables the state variable or function to be accessed from within the contract and also from external contracts or clients.
internal	Default for variables. The contract and imported/inherited contracts can access the variable or function.
private	Only from within the calling contract itself can the variable/function be accessed.
constant	Only applies to state variables. Set state variables to a value that isn't coming from storage or blockchain.
external	Only applies to functions. Can be accessed by external contracts or clients, but not from within the contract.

From previous example:

```
function validateLimit(uint256 _amount) private
returns (bool) {
  if (_amount < maxLimit)
    return true;
  return false;
}
```

```
function createCoin (address _recipient,
  uint256 _createdAmount)
  onlyOwner public {
    coinBalance[_recipient] += _createdAmount;
    emit Transfer(owner, _recipient, _createdAmount);
```

FUNCTIONS

Steven Walbr...

Function inputs. Variables sent to a function

```
function createCoin(address _recipient, uint256 _createdAmount)
```

Payable functions. Allow function to receive Ether. Revert if the function evaluates to false.

```
function getCoinValue(string _coinName)
  payable returns (uint _coinValue){
  if (msg.value > 300)
  else
    revert();
  }
```

Receive functions. A minimal/unnamed payable function with no input or outputs. Used when client called by send() or transfer() with Ether. If no such function exists, but a payable fallback function exists, the fallback function will be called ***

```
receive () external payable {
    buyTokens(_msgSender());
}
```

Function outputs. Values sent by the function to the caller.

```
function x(){
  ...
  returns (int _returnval)
}
```

Fallback functions. A minimal/unnamed payable (or not) function with no input or outputs. Used when client call doesn't match any functions or is called by send() or transfer() with Ether and no receive() function exist.. ***
**Must not run out of gas at risk of loss of Ether.

```
contract sansCoinFallback {
  function () payable {}
}
```

FUNCTION MODIFIERS

Steven Walbr...

Modifiers added to a function allow pre/post checking to allow execution when specific conditions are met.

Function modifier definition

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

Set variable used in modifier

```
constructor(uint256 _initialSupply) public {
  owner = msg.sender;
  createCoin(owner, _initialSupply);
}
```

Function with associated modifier

```
function createCoin (address _recipient,
  uint256 _createdAmount)
  onlyOwner public {
    coinBalance[_recipient] += _createdAmount;
    emit Transfer(owner, _recipient, _createdAmount);
  }
```

EVENTS

Steven Walbr...

Events are used to have a contract notify another contract or a client that something has happened.
Events are created with the keyword `event` and declared with the keyword `emit`.

```
event Transfer(address indexed from,
  address indexed to, uint256 value);
```

```
function createCoin (address _recipient,
  uint256 _createdAmount)
  onlyOwner public {
    coinBalance[_recipient] += _createdAmount;
    emit Transfer(owner, _recipient, _createdAmount);
  }
```

STORAGE

Steven Walbr...

Permanent data on the blockchain.

Other than dynamic arrays and mappings, data is stored in order starting with the first state variable placed in slot 0.
For each variable, a size measured in bytes is determined according to the variable type. `bytes` and `string` are encoded the same and stored using a `keccak256` hash of that slot's position.

```
contract SANS { uint x; }
from source unit fileX
```

```
{
    "astId": 2,
    "contract": "fileX:SANS",
    "label": "x",
    "offset": 0,
    "slot": "0",
    "type": "t_uint256"
}
```

astId: the id of the node of the state variable's declaration

contract: is the name of the contract with the path as a prefix

label: is the name of the state variable

offset: the offset in bytes within the storage

slot: storage slot where the state variable begins.

type: identifier used as a key to the variable's type info

MEMORY

Steven Walbr...

Temporary data on the blockchain.

Variables in memory arrays always occupy multiples of 32 bytes, and Solidity reserves four slots.
Solidity will place new objects at the free memory pointer (initially pointing to 0x80), and memory is never freed.

- `0x00` - `0x3f` (64 bytes): scratch space for hashing methods
- `0x40` - `0x5f` (32 bytes): currently allocated memory size (aka. free memory pointer)
- `0x60` - `0x7f` (32 bytes): zero slot

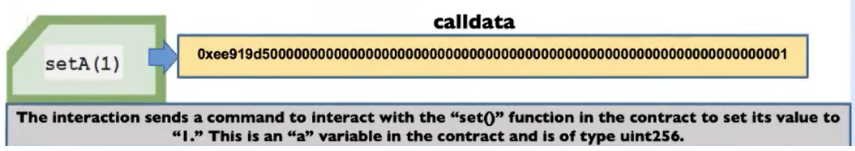
CALLDATA

Steven Walbr...

Input Data sent to a smart contract function.

The data is sent in the format defined by the Contract Application Binary Interface (ABI Specification). The ABI is the standard way to interact with contracts on EVM. Each specific function call to a contract, either from a user or another contract.

calldata with the ABI Specification is used from both the outside of blockchain, and for contract-to-contract interactions.
Because the encoding is open, not self-describing, and depends on the interface types at compile time, the **calldata** requires a schema in order to decode.



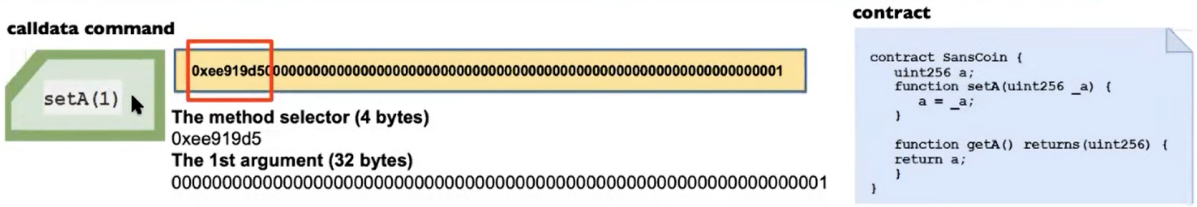

```
contract SansCoin {
   uint256 a;
   function setA(uint256 _a) {
      a = _a;
   }

   function getA() returns(uint256) {
   return a;
   }
}
```

4-BYTE SELECTOR (1)

Steven Walbr...

The hex value making up the **calldata** begins with the 4-byte **Keccak-256** hash of that function's signature. Once the function call is made, it is executed by deriving and matching up the bytecode with the corresponding 4-byte address using the contract dispatcher. Any bytes in the **calldata** after this 4-byte value represent the function's parameters and are represented with 32 bytes with padding.

The first four bytes are the method selector. The method selector is the keccak256 hash of the method signature; in this example, it is setA(uint256).



CONTRACT INTERACTION

Steven Walbr...

When a smart contract has been compiled and deployed to a blockchain, it is given a unique 42-character hex consisting of numbers and uppercase and/or lowercase letters (e.g., 0x62a34C55...).
This is exactly the same as addresses and allows users to look up transactions on the ledger that this address was involved with, such as Etherscan.

They also expose their ABI to provide methods for interacting. The interactions are similar to REST APIs, since they usually take encoded JSON, and perform similar to GET/POST type HTTP Methods.

These interactions are the same as sending/receiving Ether but can also be more complex depending on the purpose of the contract.

The most important part of the contract is its "**state**." This enables a contract to hold Ether. The contract can be loaded with Ether at deployment or sent from one address to another or to an individual user. It all depends on the contract's business logic.

WEB3

Steven Walbr...

All interactions communicate to blockchain with a protocol called **web3**.

Web3 communicates and translates the function calls and can utilize and interact with ABIs by sending the bytecode.

There are multiple libraries and SDKs for developing, initiating, and sending interactions via web3, as long as it can easily convert the desired function into its 4-byte equivalent with any parameters and call the relevant function that matches a contract section. Some options are:

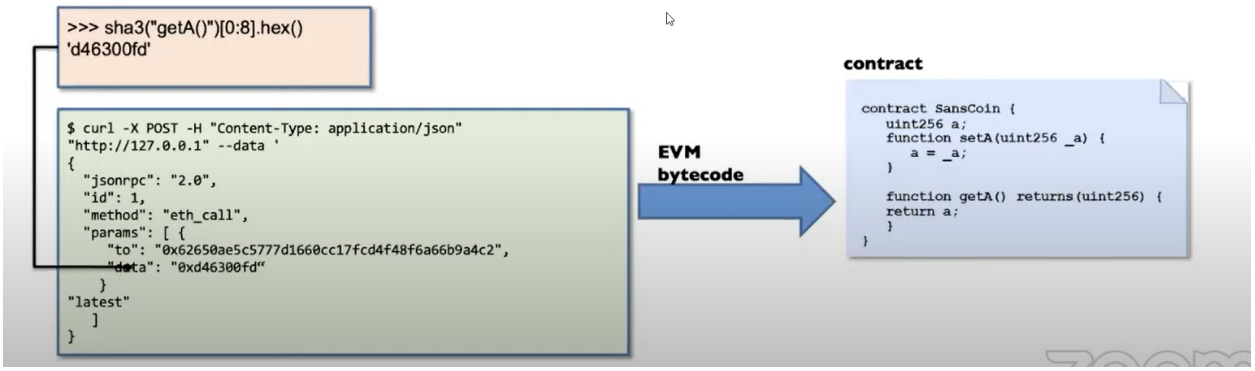
- **curl**
- **npm/web3js**
- **openzeppelin**
- **Py-EVM**
- **REMIX**
- **metamask**

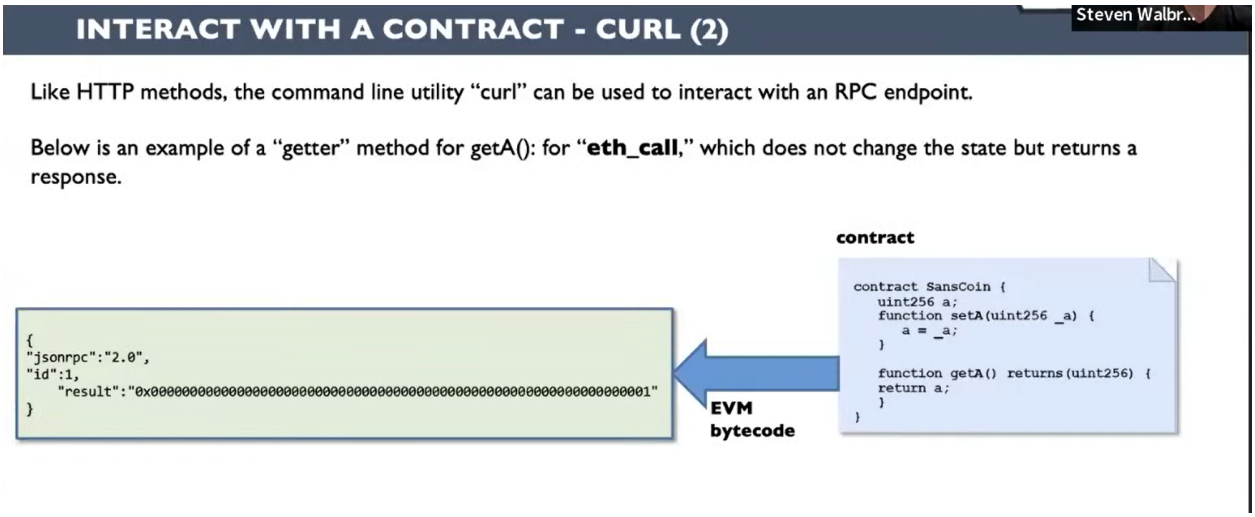
INTERACT WITH A CONTRACT - CURL (1)

Steven Walbr...

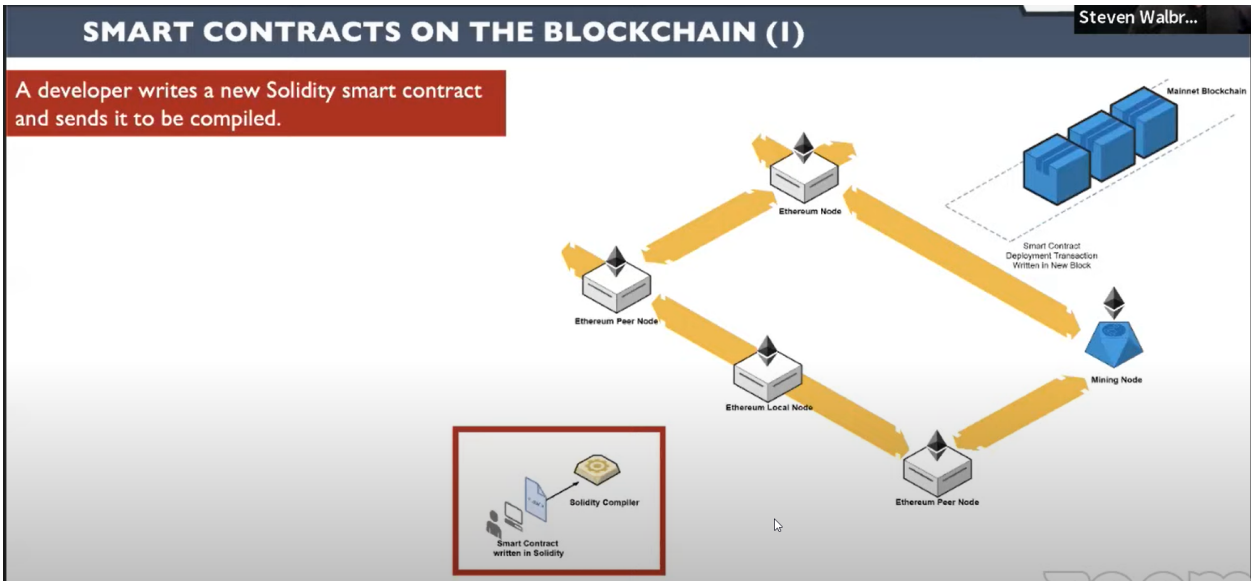
Like HTTP methods, the command line utility "curl" can be used to interact with an RPC endpoint.

Below is an example of a "getter" method for getA(): for "**eth_call**," which does not change the state but returns a response.





Working of smart contract

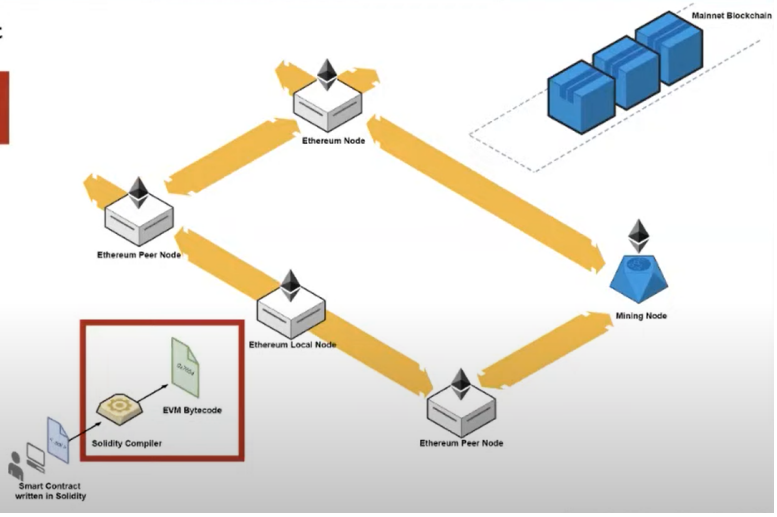


SMART CONTRACTS ON THE BLOCKCHAIN (2)

Steven Walbr...

A developer writes a new Solidity smart contract and sends it to be compiled.

The smart contract is compiled into EVM bytecode.



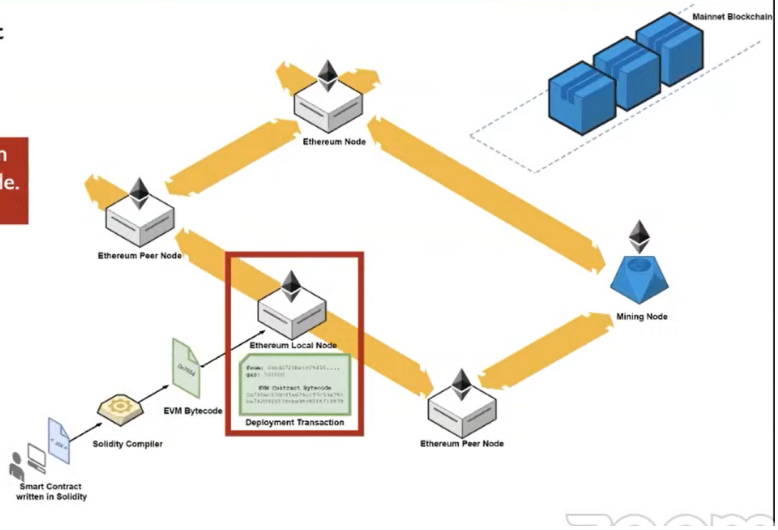
SMART CONTRACTS ON THE BLOCKCHAIN (3)

Steven Walbr...

A developer writes a new Solidity smart contract and sends it to be compiled.

The smart contract is compiled into EVM bytecode.

Once in EVM bytecode, a deployment transaction is issued and sends the information to a local node.



SMART CONTRACTS ON THE BLOCKCHAIN (4)

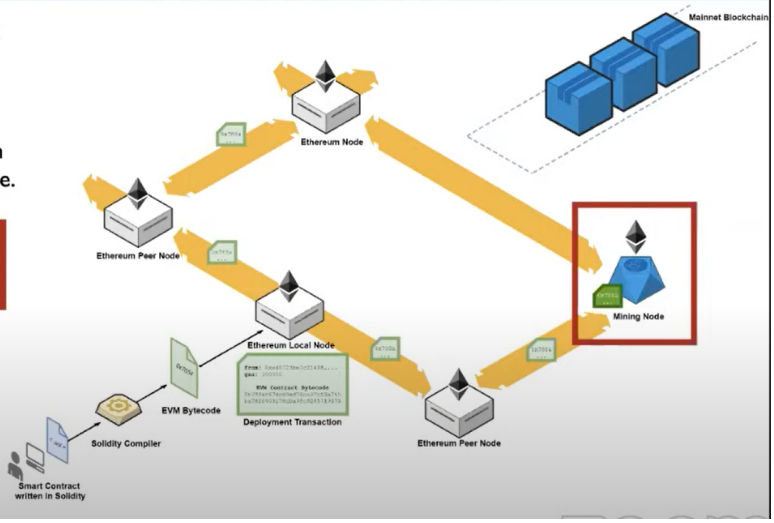
Steven Walbr...

A developer writes a new Solidity smart contract and sends it to be compiled.

The smart contract is compiled into EVM bytecode.

Once in EVM bytecode, a deployment transaction is issued and sends the information to a local node.

The deployment transaction (and the smart contract code) is propagated to peer nodes and outward until it is received by a mining node.



SMART CONTRACTS ON THE BLOCKCHAIN (5)

Steven Walbr...

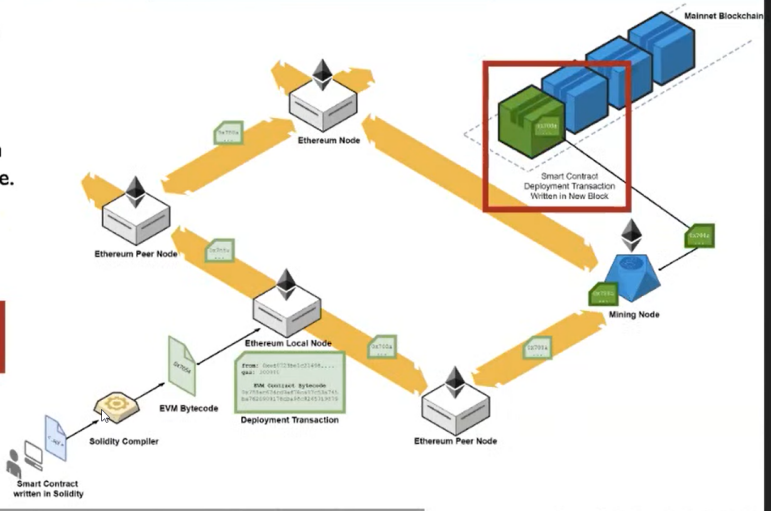
A developer writes a new Solidity smart contract and sends it to be compiled.

The smart contract is compiled into EVM bytecode.

Once in EVM bytecode, a deployment transaction is issued and sends the information to a local node.

The deployment transaction (and the smart contract code) is propagated to peer nodes and outward until it is received by a mining node.

The contract EVM bytecode is processed by the mining node and written to the next new block.



SMART CONTRACTS ON THE BLOCKCHAIN (6)

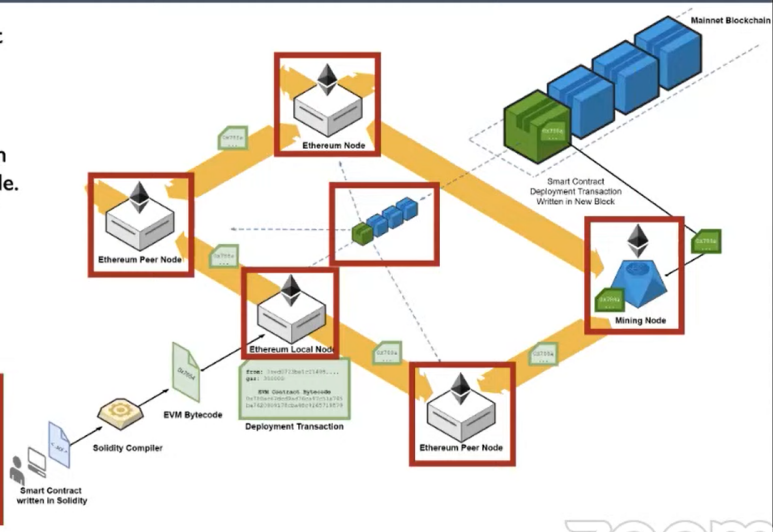
Steven Walbr...

A developer writes a new Solidity smart contract and sends it to be compiled.
The smart contract is compiled into EVM bytecode.
Once in EVM bytecode, a deployment transaction is issued and sends the information to a local node.

The deployment transaction (and the smart contract code) is propagated to peer nodes and outward until it is received by a mining node.

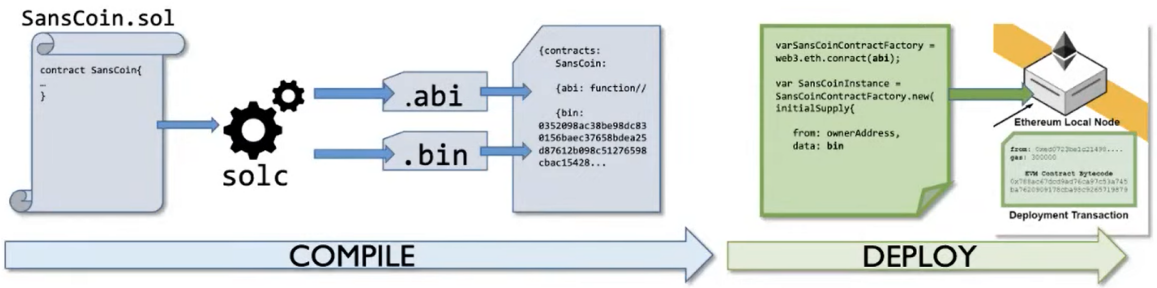
The contract EVM bytecode is processed by the mining node and written to the next new block.

The contract deployment transaction is finally written to the blockchain just like any other Ether or data transaction and then replicated to all nodes in the Ethereum network, achieving consensus, and can be called or viewed.



HIGH-LEVEL COMPILING AND DEPLOYMENT PROCESS

Steven Walbr...



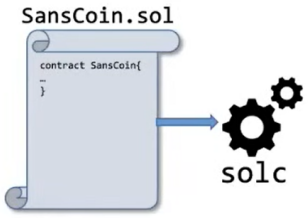
Contract compiled with solc

Contract bytecode files created

Contract bytecode files deployed to blockchain

COMPILER - SOLC

Steven Walbr...



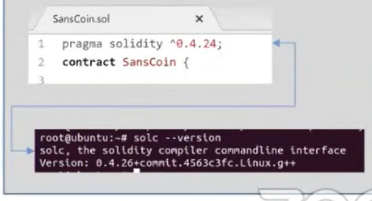

```
zilon@zilon-SEC554:~$ solc --help
solc, the Solidity commandline compiler.

This program comes with ABSOLUTELY NO WARRANTY. This is free software, and you
are welcome to redistribute it under certain conditions. See 'solc --license'
for details.

Usage: solc [options] [input_file...]
Compiles the given Solidity input files (or the standard input if none given or
"-" is used as a file name) and outputs the components specified in the options
at standard output or in files in the output directory, if specified.
Imports are automatically read from the filesystem, but it is also possible to
remap paths using the context:prefix=path syntax.
Example:
    solc --bin -o /tmp/solcoutput dapp-bin=/usr/local/lib/dapp-bin contract.sol
```

The solc compiler version and the pragma version must match to successfully compile.

There is a tool, solc-select, that allows you to easily switch the version to match the contract.

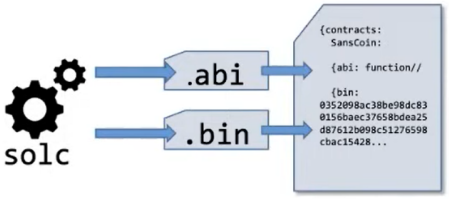
Invoke this with:

```
solc-select <version>
```



ABI (APPLICATION BINARY INTERFACE) AND EVM BYTECODE

Steven Walbr...



.abi – This file displays the API that the contract exposes and the functions that clients use.

.bin – This contracts EVM bytecode used in a deployment transaction and is mined by an Ethereum node to be added to the blockchain

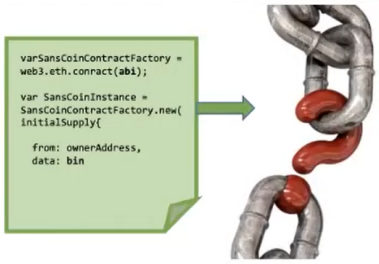
ETHEREUM NETWORKS

Steven Walbr...

Before deploying a contract to the mainnet, it should be tested and validated.

Test networks allow a user to validate the contract is secured (or vulnerable) and use fake Ethereum from a "faucet."

Remember, once a contract is deployed to mainnet, it's publicly available on the blockchain and can't be changed or patched.



Ethereum Networks	Description
Mainnet	The Production Ethereum network, based on Proof of Work.
Ropsten	Mirror Image of Main network, but for testing. Ethereum is mined easily.
Rinkeby	Test network, but with Proof of Authority instead of Proof of Work. (geth)
Kovan	Test network, but with Proof of Authority instead of Proof of Work. (parity)
Ganache-CLI and Ganache-UI (tool)	A "personal blockchain" that can be locally created and hosted for testing.

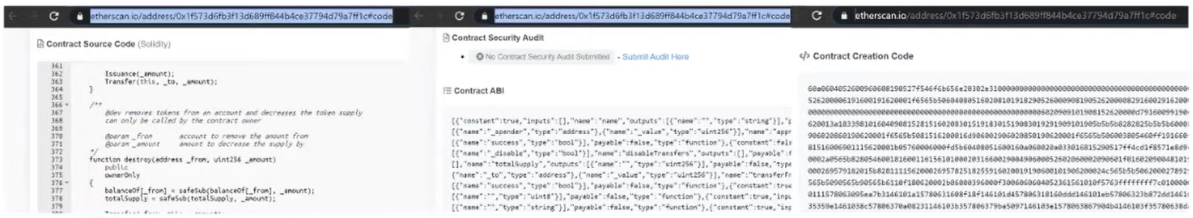
ETHERSCAN

Steven Walbr...

Etherscan is the online resource to view all the transactions, contracts, addresses, and underlying components for the entire Ethereum blockchain.

Here is an example of a deployed smart contract in production for Bancor's SmartToken contract.

You can also browse here to view it yourself. https://etherscan.io/address/0x1f573d6fb3f13d689ff844b4ce37794d79a7ff1c#code

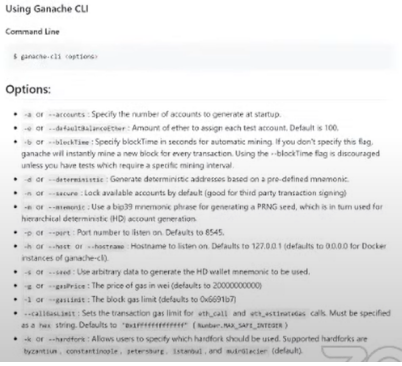


Solidity Code ABI Bytecode

GANACHE-CLI

Steven Walbr...

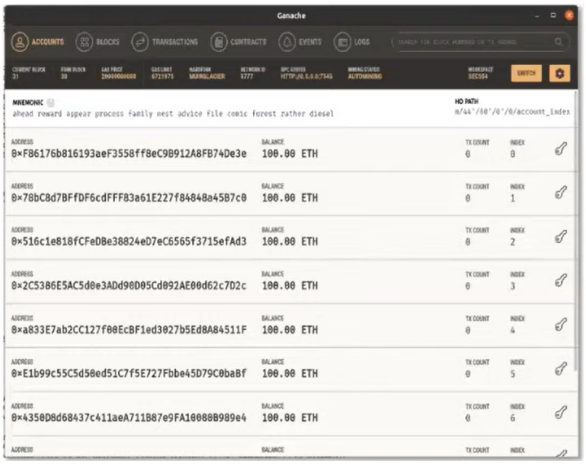
Ganache-CLI is a personal blockchain tool used for Ethereum testing and development.
It uses ethereumjs to simulate full client behavior and makes developing smart contracts faster and easier. It includes all popular RPC functions and features (like events) and can be run deterministically. It's also good for deploying vulnerable contracts to POC and demonstrating vulnerabilities and exploits.



GANACHE-UI

Steven Walbr...

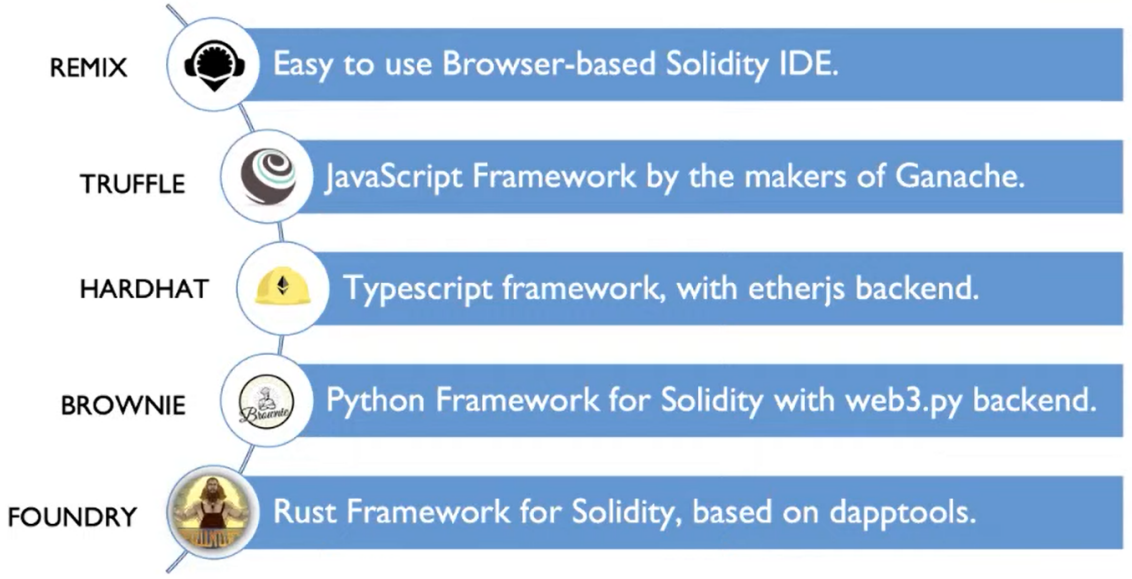
Ganache-UI is the same personal blockchain tool as Ganache-CLI, but it is a desktop application with a full GUI.



A personal Ethereum blockchain can be created locally that allows options to set gas limits, and gas price, fork blocks, deploy and interact with contracts, and create test addresses with Ether allocated.

It listens for RPC commands on localhost port 7454 by default.

DEVELOPMENT AND AUDITING FRAMEWORKS



TRUFFLE

A development/testing framework for EVM, Truffle is a full suite with integrations and easy deployment. Truffle can make many developers' and testers' lives easier. In fact, it can take care of the compilation part we just did manually with solc.

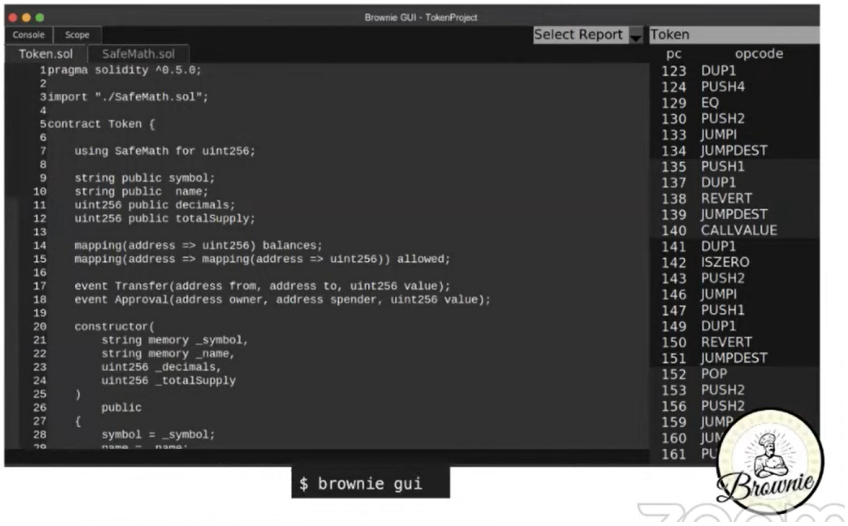
Truffle provides:

- Built-in smart contract compilation, linking, deployment, and binary management
- Automated contract testing for rapid development
- Scriptable, extensible deployment, and migrations framework
- Network management for deploying to any number of public and private networks
- Package management with EthPM and NPM, using the ERC-190 standard
- Interactive console for direct contract communication
- Configurable build pipeline with support for tight integration
- External script runner that executes scripts within a Truffle environment
- Direct integration with "Ganache" and "Ganache-UI" from the previous exercises

BROWNIE

Steven Walbr...

- Uses *web3.py* on backend.
- Contract testing via "*pytest*" including trace-based evaluation.
- Property-based and stateful testing via "*hypothesis*".
- Python style debugging, with tracebacks and custom error strings.
- Built-in CLI and local network setup, as well as GUI.



TRUFFLE DEPLOYMENT

Steven Walbr...

`truffle init`

Creates a bare Truffle project with no smart contracts included.

This sets up the directory structure needed:

```
contracts/      -- directory for Solidity files
migrations/     -- directory for script deployment files
test/           -- directory for Solidity files
truffle-config.js – configuration and settings
```

`truffle compile`

Compiles the Solidity files in the contracts folder and places the outputs and bytecode in:

```
build/contracts/
```

`truffle migrate`

This will run the deployment scripts and send the contract to a blockchain or Ganache (as specified in the config file). The migration scripts are in the migrations folder.

```
var MyContract = artifacts.require("MyContract");

module.exports = function(deployer) {
  // deployment steps
  deployer.deploy(MyContract);
};
```

example migration script

BROWNIE DEPLOYMENT

Steven Walbr...

- An example Brownie script, which would be in:
  scripts/SANStoken.py
- Used to deploy the `SANSToken.sol` contract located in:
  contracts/SANSToken.sol

Brownie Project Structure

- `contracts/` : Contract sources
- `interfaces/` : Interface sources
- `scripts/` : Scripts for deployment and interaction
- `tests/` : Scripts for testing the project
- `build/` : Project data such as compiler artifacts and unit test results
- `reports/` : JSON report files for use in the GUI

SANStoken.py

```python
from brownie import SANSToken, accounts

def main():
    token = SANSToken.deploy("SANS Token", "SANS", 18, 1e23, {'from': accounts[0]})
    return token

def distribute_tokens(sender=accounts[0], receiver_list=accounts[1:]):
    token = main()
    for receiver in receiver_list:
        token.transfer(receiver, 1e18, {'from': sender})
```

Step 1: `$ brownie compile`

```
halborn@stevens-mbp token % brownie compile
Brownie v1.17.0 - Python development framework for Ethereum

Compiling contracts...
  Solc version: 0.6.12
  Optimizer: Enabled  Runs: 200
  EVM Version: Istanbul
Generating build data...
 - SafeMath
 - SANSToken

Project has been compiled. Build artifacts saved at /Users/halborn/Downloads/brownie/token/build/contracts
halborn@stevens-mbp token %
```

Step 2: `$ brownie run scripts/<YOUR_PYTHON_DEPLOY_SCRIPT.py>`

```
halborn@stevens-mbp token % brownie run scripts/SANStoken.py
Brownie v1.17.0 - Python development framework for Ethereum

TokenProject is the active project.

Launching 'ganache-cli --port 8545 --gasLimit 12000000 --accounts 10 --hardfork istanbul
 --mnemonic brownie'...

Running 'scripts/SANStoken.py::main'...
Transaction sent: 0x736f6702dbca0d035e0739cb57c5c534da4dfe238acad8e4fec688590b477a8a
  Gas price: 0.0 gwei   Gas limit: 12000000   Nonce: 0
  SANSToken.constructor confirmed   Block: 1   Gas used: 512517 (4.27%)
  SANSToken deployed at: 0x3194cBDC3dbcd3E11a07892e7bA5c3394048Cc87

Terminating local RPC client...
halborn@stevens-mbp token %
```

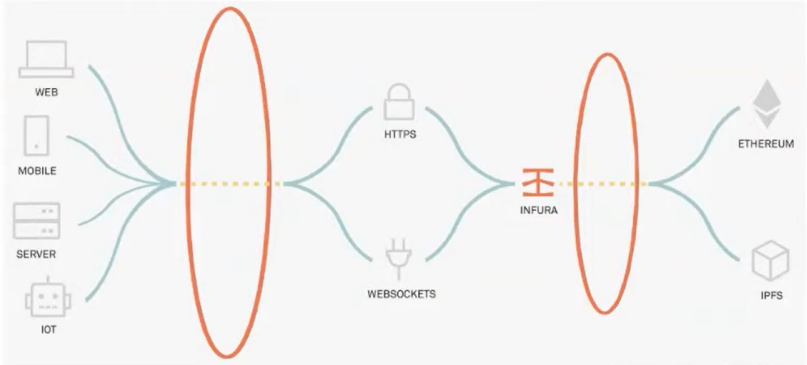
INFURA

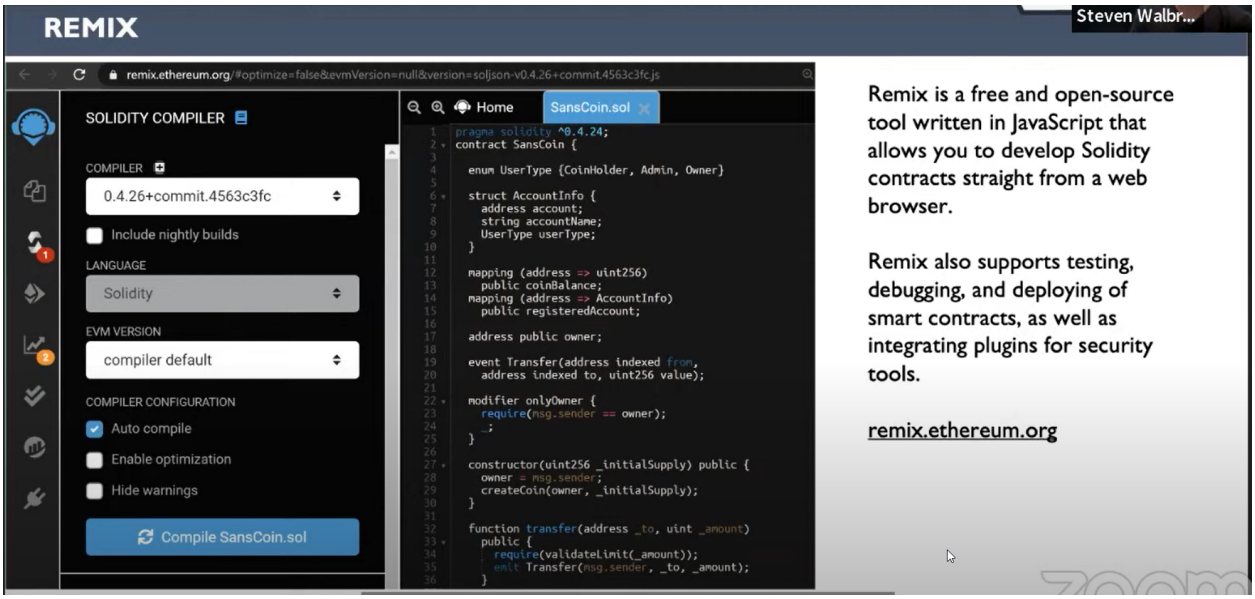
Steven Walbr...

Infura is an API access provider and development suite for the Ethereum networks focused on blockchain infrastructure. With an Infura account, you can connect to an Ethereum blockchain without having to host your own. Users can also "fork" from a specific block.

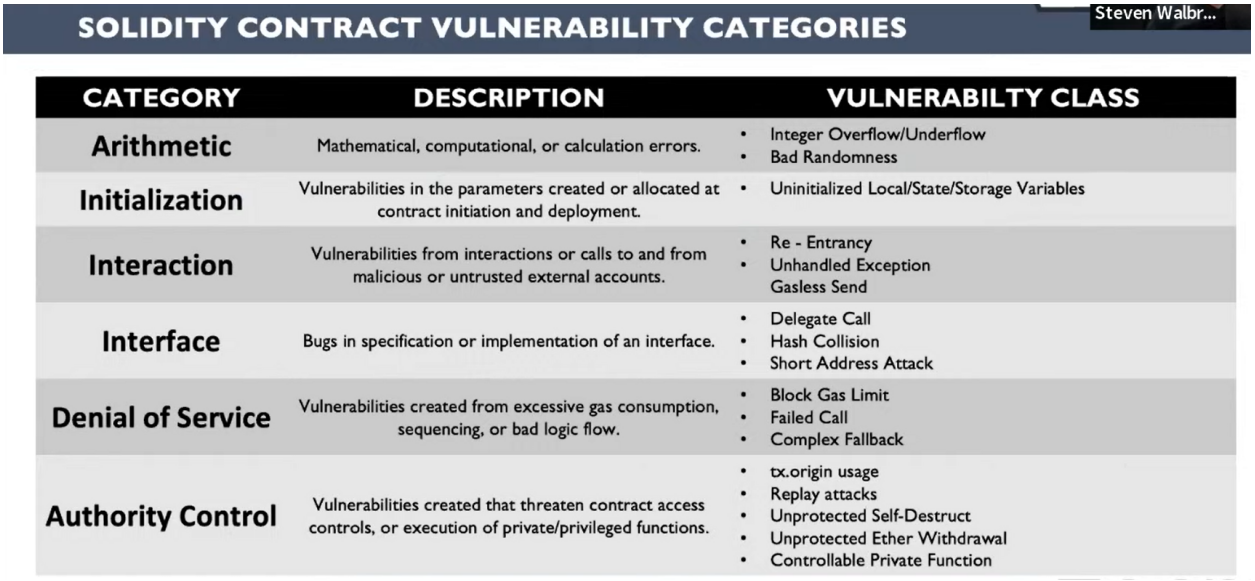
https://infura.io

- Free and paid versions
- Same tech behind MetaMask nodeless connections
- Communicates over HTTPS or WebSocket (WSS)
- Hosts various Ethereum networks





VULNERBILTIY



CATEGORY	DESCRIPTION	VULNERABILTY CLASS
Arithmetic	Mathematical, computational, or calculation errors.	• Integer Overflow/Underflow • Bad Randomness
Initialization	Vulnerabilities in the parameters created or allocated at contract initiation and deployment.	• Uninitialized Local/State/Storage Variables
Interaction	Vulnerabilities from interactions or calls to and from malicious or untrusted external accounts.	• Re - Entrancy • Unhandled Exception Gasless Send
Interface	Bugs in specification or implementation of an interface.	• Delegate Call • Hash Collision • Short Address Attack
Denial of Service	Vulnerabilities created from excessive gas consumption, sequencing, or bad logic flow.	• Block Gas Limit • Failed Call • Complex Fallback
Authority Control	Vulnerabilities created that threaten contract access controls, or execution of private/privileged functions.	• tx.origin usage • Replay attacks • Unprotected Self-Destruct • Unprotected Ether Withdrawal • Controllable Private Function

REENTRANCY/RECUSIVE CALL ATTACKS

Steven Walbr...

One of the major dangers of calling external contracts is that they can take over the control flow. In the reentrancy attack (aka recursive call attack), a malicious contract calls back into the calling contract before the first invocation of the function is finished. A vulnerability can often be exploited when internal state changes (like Ethereum balances) are not updated before the call is executed. (aka "Checks-Effects-Interactions pattern")

Vulnerable

```
pragma solidity 0.4.24;

contract SimpleDAO {
  mapping (address => uint) public credit;

  function donate(address to) payable public{
    credit[to] += msg.value;
  }

  function withdraw(uint amount) public{
    if (credit[msg.sender]>= amount) {
      require(msg.sender.call.value(amount)());
      credit[msg.sender]-=amount;
    }
  }

  function queryCredit(address to) view public returns(uint){
    return credit[to];
  }
}
```

state change after external call

Not-Vulnerable

```
pragma solidity 0.4.24;

contract SimpleDAO {
  mapping (address => uint) public credit;

  function donate(address to) payable public{
    credit[to] += msg.value;
  }

  function withdraw(uint amount) public {
    if (credit[msg.sender]>= amount) {
      credit[msg.sender]-=amount;
      require(msg.sender.call.value(amount)());
    }
  }

  function queryCredit(address to) view public returns (uint){
    return credit[to];
  }
}
```

INTEGER OVERFLOW/UNDERFLOW (ARITHMETIC ATTACKS)

Steven Walbr...

This is the buffer overflow of smart contracts. An integer overflow or underflow happens when an arithmetic operation reaches the maximum or minimum size of a type. If a number is stored in the uint8 type, then it is stored in an 8-bit unsigned number ranging from 0 to 2^8-1.

The vulnerability occurs when an arithmetic operation attempts to create a numeric value that is outside of the range possible by the type, either larger than the maximum (overflow) or lower than the minimum (underflow) representable value.

Vulnerable

```
pragma solidity ^0.4.19;
contract IntegerOverflow {
    uint public count = 1;
function run(uint256 input) public {
        count -= input;
    }
}
```

uint type overflow is checked with SafeMath library function call

Not Vulnerable

```
pragma solidity ^0.4.19;
contract IntegerOverflow {
    uint public count = 1;
function run(uint256 input) public {
        count = sub(count,input);
    }
//from SafeMath
    function sub(uint256 a, uint256 b) internal pure returns (uint256) {
        require(b <= a);//SafeMath uses assert here
        return a - b;
    }
}
```

DENIAL OF SERVICE WITH BLOCK GAS LIMIT

Steven Walbr...

Execution of the functions inside of a smart contract always requires a certain amount of gas. The gas amount is determined by how much computation is needed to complete a transaction. The Ethereum network specifies a threshold for each block that is mined, and the sum of all transactions included in a block can not exceed that limit.

Vulnerable

```
1  pragma solidity ^0.4.25;
2  contract DosFunction {
3  address[] listAddresses;
4  function infiniteArray() public returns (bool){
5          if(listAddresses.length<1500) {
6  for(uint i=0;i<350;i++) {
7                  listAddresses.push(msg.sender);
8              }
9              return true;
10 } else {
11             listAddresses = new address[](0);
12             return false;
13         }
14     }
15 }
```

This example of a denial of service is due to potentially hitting the block gas limit. The function would create an array that continuously pushes items into the address array, using up more gas each iteration and consuming most of the block gas limit. This can stop other transactions from being mined.

Note that this would become very expensive for someone to maintain, since the gas has to be spent in Ethereum by the caller.

UNPROTECTED SELFDESTRUCT

Steven W

A self-destruct instruction (or "suicide" function) is used to terminate a contract by making it unreachable from any further access or calls. Due to missing or insufficient access controls, malicious parties can self-destruct a contract and sometimes have all the Ethereum contained in the contract be transferred to an address of their choosing prior to termination.

This contract can be killed by anyone.

```
1  pragma solidity ^0.4.23;
2  contract killMe {
3      uint256 private initialized = 0;
4      uint256 public count = 1;
5  function init() public {
6          initialized = 1;
7      }
8  function run(uint256 input) {
9          if (initialized == 0) {
10             return;
11         }
12 selfdestruct(msg.sender);
13     }
14 }
```

This contract requires two parties to self-destruct.

```
1  pragma solidity ^0.4.23;
2  contract killMeSoftly {
3      uint256 private initialized = 0;
4      uint256 public count = 1;
5  function init() public {
6          initialized = 1;
7      }
8  function run(uint256 input) {
9          if (initialized != 2) {
10             return;
11         }
12 selfdestruct(msg.sender);
13     }
14 }
```

TIMESTAMP MANIPULATION

Steven Walbr...

Certain contracts, such as timed token sales, require checking the current date/time. Global namespace variables, such as block.timestamp and block.number, can be leveraged to determine the time but are not safe from external manipulation.

This contract shouldn't trust block.timestamp.

```
1  pragma solidity ^0.5.0;
2  contract TimedSale {
3  event Finished();
4    event notFinished();
5  // Sale should end on January 1, 2019
6    function isSaleFinished() private returns (bool) {
7      return block.timestamp >= 1546300800;
8    }
9  function run() public {
10     if (isSaleFinished()) {
11         emit Finished();
12     } else {
13         emit notFinished();
14     }
15   }
16 }
```

Malicious miners can alter the timestamp of their blocks. Even though miners can't set a timestamp smaller than the previous one (otherwise the block will be rejected), they can set a timestamp slightly farther ahead in the future, which can sometimes provide a "first mover advantage" on certain contracts.

DELEGATECALL

Steven Walbr...

Delegatecall is like a message call, except the code at the target address is executed in the context of the calling contract. msg.sender and msg.value do not change their values, which allows a smart contract to dynamically load code from a different address during execution. The ether balance, the current address, and storage still refer to the calling contract.

Calling into untrusted contracts is an opportunity for exploitation since the code at the target address has full control over the caller's balance and the values in its storage.

Vulnerable

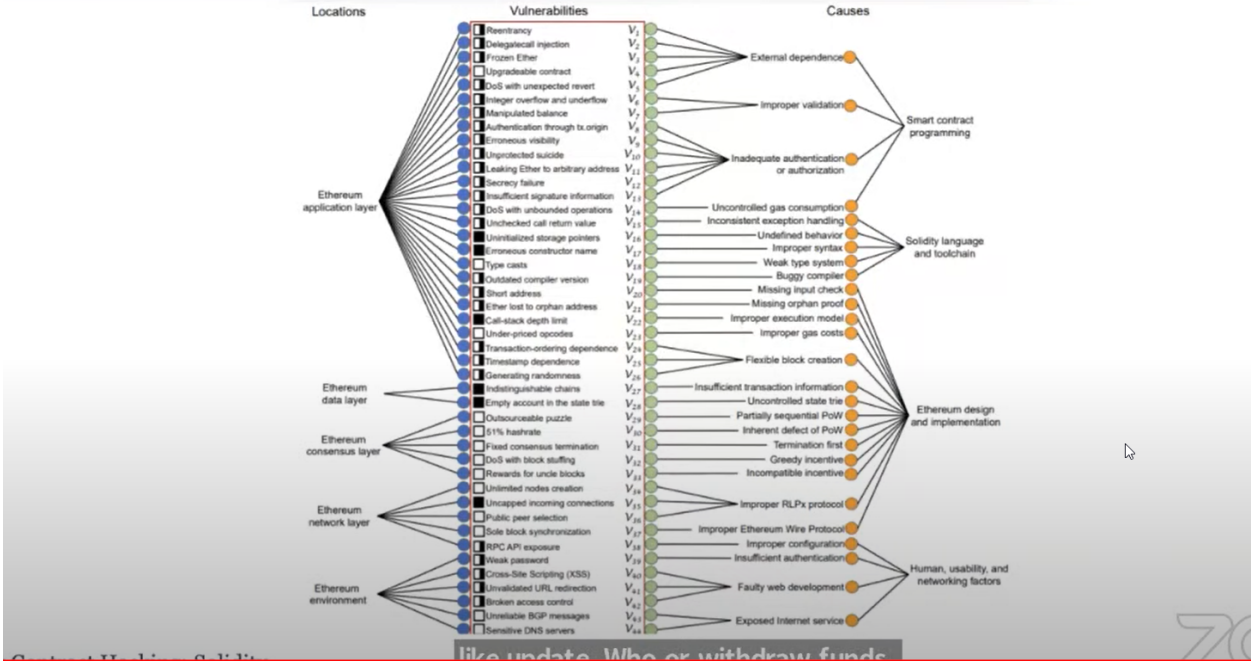
```
1  pragma solidity ^0.4.24;
2  contract Proxy {
3  address owner;
4  constructor() public {
5      owner = msg.sender;
6    }
7  function forward(address callee, bytes _data) public {
8      require(callee.delegatecall(_data));
9    }
10 }
```

The fixed contract uses modifiers and sets boundaries on the functions of the caller and callee.

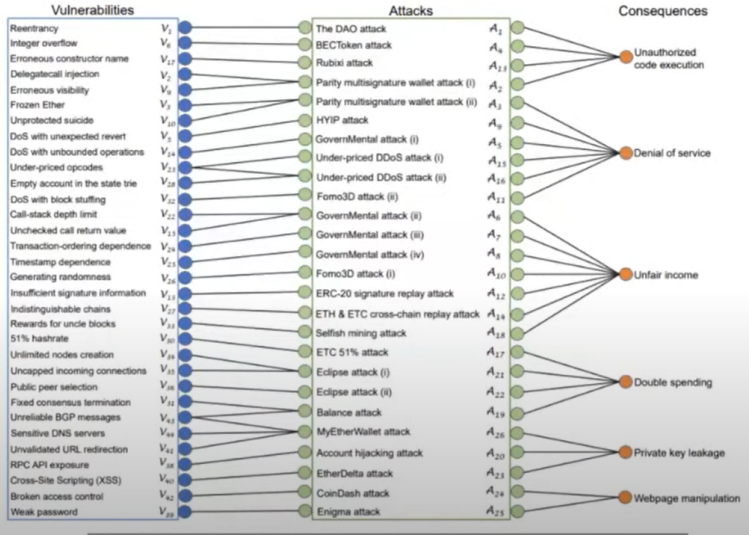
Not Vulnerable

```
1  pragma solidity ^0.4.24;
2  contract Proxy {
3  address callee;
4    address owner;
5  modifier onlyOwner {
6      require(msg.sender == owner);
7      _;
8    }
9  constructor() public {
10     callee = address(0x0);
11     owner = msg.sender;
12   }
13 function setCallee(address newCallee) public onlyOwner {
14     callee = newCallee;
15   }
16 function forward(bytes _data) public {
17     require(callee.delegatecall(_data));
18   }
19 }
```

INFOGRAPHIC OF VULNERABILITY CLASSES AND CAUSES



INFOGRAPHIC OF VULNERABILITY ATTACKS AND CONSEQUENCES



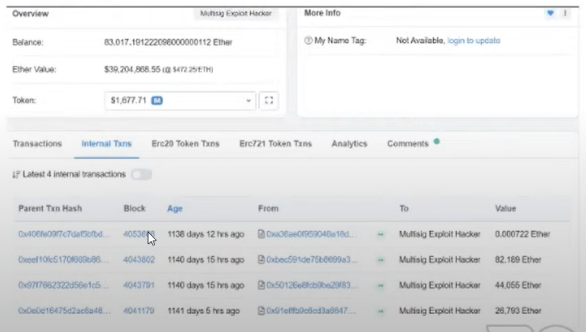
THE PARITY BUG (1)

Steven Walbr...

An Ethereum multi-sig wallet had a bug in the code that allowed an attacker to take ownership of a wallet in a single transaction and drain it of its funds.

The MultisigExploit-Hacker (MEH), as he or she is named, exploited a vulnerability with the **delegatecall** issue explained previously and was able to make off with $30 million worth of Ethereum.

https://etherscan.io/address/0xb3764761e297d6f121e79c32a65829cd1ddb4d32#internaltx



THE PARITY BUG (2)

Steven Walbr...

- The Parity wallet bug was a prominent vulnerability on the Ethereum blockchain, which caused 280 million USD worth of Ethereum to be frozen on the Parity wallet account.
- It was due to a very simple vulnerability: a library contract used by the Parity wallet was not initialized correctly and could be owned or destructed by anyone.
- Once the library was destructed, any call to the wallet library would then fail, effectively locking all funds.

The user "devops199" accidently called self-destruct, locking all money in the contract forever.

THE PARITY BUG (3)

Steven Walbr...

The `initWallet()` function is used to set up the initial state whenever a new multi-sig wallet is created.

The `kill()` function is used to self-destruct.

Every Parity multi-sig wallet that was created up to July 20, 2017, relied on this library contract.

```
// 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);
}
```

THE PARITY BUG (4)

Steven Walbr...

On Monday, November 6, 2017, a vulnerability in the "library" contract code deployed as a shared component of all Parity multi-sig wallets that were deployed was found by an anonymous user. The first exploit was a transaction where a user called the init() function on the library contract and provided their own address as the owner. The user decided to exploit this vulnerability and made himself the "owner" of the library contract. This effectively turned the contract into a wallet instead of a library.

https://etherscan.io/address/0x863df6bfa4469f3ead0be8f9f2aae51c91a907b4

0.0019135336 Ether ($0.63)

3444021

Success

88

```
Function: initWallet(address[] _owners, uint256 _required, uint256 _daylimit)

MethodID: 0xe46dcfeb
[0]:0000000000000000000000000000000000000000000000000000000000000060
[1]:0000000000000000000000000000000000000000000000000000000000000000
[2]:0000000000000000000000000000000000000000000000000000000000000000
[3]:0000000000000000000000000000000000000000000000000000000000000001
[4]:000000000000000000000000ae7168deb525862f4fee37d987a971b385b96952
```

Convert To Ascii

Init function called that turned the wallet library into a single owner wallet.

THE PARITY BUG (5)

Steven Walbr...

Subsequently, the user destructed this component. Since Parity multi-signature wallets depend on this component, this action blocked funds in 587 wallets holding a total amount of 513,774.16 Ether as well as additional tokens.

0.0009395152 Ether ($0.31)

4750547

Success

89

```
Function: kill(address _to)

MethodID: 0xcbf0b0c0
[0]:0000000000000000000000000ae7168deb525862f4fee37d987a971b385b96952
```

Convert To Ascii

https://etherscan.io/GitHubaddress/0x863df6bfa4469f3ead0be8f9f2aae51c91a907b4