Developing Secure Smart Contracts
Final - OWASP Toronto
January 23, 2019
Whoami

• Jamie Baxter, M. Eng., OSCP, OSCE, CISSP, GPEN
• Independent Information Security Consultant focusing on security assessments (applications, infrastructures and smart contracts)
• Previously worked in aerospace, government and finance sectors
• CTF’er, pen-tester, red teamer, appsec
Tonight

• What is a Smart Contract?
• Ethereum Overview
• Smart Contract Introduction
• Smart Contract Vulnerabilities
• Resources And Capture The Flags
What are Ethereum Smart Contracts?

• Def: A Ethereum Smart Contract is a program that defines a general purpose computation which takes place on a blockchain or distributed ledger

• Term originally coin by Nick Szabo

• The smart contract code facilitates, verifies, and enforces the negotiation or performance of an agreement or transaction.

• While self-verifying, self-executing and tamper resistant smart contracts may contain bugs, from programmer errors to flaws in the compiler & toolchain to the platform itself.

Source:
https://blockchainhub.net/smart-contracts/
https://en.wikipedia.org/wiki/Smart_contract/
The address that triggered the Parity bug. The event was reported on this Github ticket.
Ethereum is a Transaction Based State Machine

A transaction is a single cryptographically-signed instruction
What is a World State ($\sigma$)?

- It is the mapping between addresses and their account state at a given time.
What’s in an Account?
There’s actually two types of accounts

Externally Owned Accounts (EOA)

Address($\alpha_1$) → Account State ($\sigma[\alpha_1]_n$)
- Nonce
- Ether Balance

Contract Account

Address($\alpha_2$) → Account State ($\sigma[\alpha_2]_n$)
- Nonce
- Ether Balance
- Code Hash
- Storage Hash

Code
Storage
A Word on Addresses

Externally Owned Account (EOA) Address (A)

\[ A = B_{96.255}(KEC(PUBKEY(p_r))) \]  
Where \( p_r \) is the private key

Contract Accounts Address (A)

\[ A = B_{96.255}(KEC(Sender\ Address, Nonce)) \]
Account Type Summary

Externally Owned Accounts

• Have a nonce
• Have an Ether balance
• Can send transactions
  • Transfers
  • Messages to Contracts or other EOAs
• Only EOA can initiate transactions

Contract Accounts

• Have a nonce
• Have an Ether balance
• Code hash
• Code execution is triggered by a transaction
• Can call other contracts
Multiple Transactions are Combined in a Block

Block (Bx)

- Headers
- Transaction (T₁)
- Transaction (T₂)
- Transaction (T₃)

Also Cryptographically Signed

World State $\sigma[t]$

Transition Function

Ethereum Virtual Machine EVM

World State $\sigma[t + 1]$
The Sequence of Blocks and World States

...is the Blockchain!
The Transition Function - Ethereum Virtual Machine (EVM)

- Turing complete instruction set $2^8$ Op Codes, Fixed Length
- 256-bit word machine
- 1024 element stack (of 256 bits each)
- 8-Bit opcodes
- No registers (purely stack based)
- Storage (persistent / per account)
- Memory (volatile)
- It’s purpose is run EVM Byte Code (aka Smart Contracts)
What are Ethereum Smart Contracts?

• Smart Contracts are very similar to classes in C++ or Java
• All Smart Contracts are bound to an address and have an ether balance associated with them
• Smart Contracts have a constructor (no overloading though)
• Solidity supports inheritance and polymorphism
• Other objected orientated concepts like visibility (private, public), state variables and interfaces also all apply
• Compiled to EVM Bytecode and stored in the world state indexed by code hash
• Contracts can be killed (suicide)
• Usually written in Solidity. But other languages exist ex: LLL
Life Cycle of a Smart Contract

Transaction to Create
• Issued by a EOA or another Smart Contract (contracts can create contracts)

Execution Driven by Transactions
• Receive transactions (calls, delegate calls)
• Perform actions
• Functions called from other functions

Suicide or “Freeze”

Every Contract is stored within the world state.
Contract Execution - Everything has a Price!

- Cost is measured in “GAS”
- The unit price of GAS in Ether is defined by the initiator of the transaction.
- Creating a contract costs GAS
- All execution steps cost GAS
- The more complex the execution the greater the cost
- Each transaction is provided a GAS stipend to begin execution
- Each block is subject to the GAS limit of 8 million.
  - Consider an expensive transaction like SSTORE (20000 Gas) means a block can write to store 400 times
  - Ethereum network can process about 25 transactions per second. Though multiple initiatives are underway to greatly increase that

### Partial List of GAS costs

<table>
<thead>
<tr>
<th>Operation Name</th>
<th>Gas Cost</th>
<th>Remark</th>
</tr>
</thead>
<tbody>
<tr>
<td>step</td>
<td>1</td>
<td>default amount per execution cycle</td>
</tr>
<tr>
<td>stop</td>
<td>0</td>
<td>free</td>
</tr>
<tr>
<td>suicide</td>
<td>0</td>
<td>free</td>
</tr>
<tr>
<td>sha3</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>sload</td>
<td>20</td>
<td>get from permanent storage</td>
</tr>
<tr>
<td>sstore</td>
<td>100</td>
<td>put into permanent storage</td>
</tr>
<tr>
<td>balance</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>create</td>
<td>100</td>
<td>contract creation</td>
</tr>
<tr>
<td>call</td>
<td>20</td>
<td>initiating a read-only call</td>
</tr>
<tr>
<td>memory</td>
<td>1</td>
<td>every additional word when expanding memory</td>
</tr>
<tr>
<td>txdata</td>
<td>5</td>
<td>every byte of data or code for a transaction</td>
</tr>
<tr>
<td>transaction</td>
<td>500</td>
<td>base fee transaction</td>
</tr>
<tr>
<td>contract creation</td>
<td>53000</td>
<td>changed in homestead from 21000</td>
</tr>
</tbody>
</table>
Distributed Applications (dApps) (Simplified)

Contract(s) Backend  Web Gui Front End
An Example dApp - CryptoKitties!

WHO WOULD WIN?

| A decentralized network comprising over 25,000 active nodes | A Neopet |
A recent Dapp Ranking

Source: http://dappradar.com
## Tools – A Sampling

<table>
<thead>
<tr>
<th>Tool</th>
<th>Descriptions</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Metamask</td>
<td>A Browser Extension for Running dApps</td>
<td>Wallet Integration</td>
</tr>
<tr>
<td>Mist</td>
<td>Dedicated Dapp Browser</td>
<td>Wallet Integration</td>
</tr>
<tr>
<td>Ganache</td>
<td>Ethereum Personal Blockchain (Now you can have a blockchain too!)</td>
<td>“Ganache is a personal blockchain for Ethereum development you can use to deploy contracts, develop your applications, and run tests”</td>
</tr>
<tr>
<td>Truffle</td>
<td>Smart Contract Development Suite</td>
<td>Compile and Deploy Smart Contracts</td>
</tr>
<tr>
<td>Remix</td>
<td>IDE</td>
<td>Online</td>
</tr>
<tr>
<td>Geth</td>
<td>Ethereum Node Controller (can join main or multiple test and special purpose nets)</td>
<td><strong>geth</strong> is the command line interface for running a full ethereum node implemented in Go.</td>
</tr>
</tbody>
</table>

Geth is the command line interface for running a full ethereum node implemented in Go.
So, of course, all the past lessons in software security have been applied and Smart Contracts are now *bug free*...

Thanks for coming out!
Everything old is new again!

- Integer Underflow / Overflow (SWC-101)
- Unprotected Sensitive Functions (Self-Destruct) (SWC-106)
- Exposed Private Data
- Bad Randomness (SWC-120)
- Re-Entrancy (SWC-107)
- Unsafe Authorization (SWC-115)
- Unsafe Contract Constructors (SWC-115)
- Out-Of-Bounds Write-Anywhere (SWC-124)
- Unprotected Withdrawal

There are currently 29 weakness patterns identified in Smart Contracts:
Source: https://en.wikipedia.org/wiki/Integer_overflow
Integer Overflows have been with us...for a long, long time!

Source: https://en.wikipedia.org/wiki/Integer_overflow
pragma solidity ^0.4.24;

contract OverflowAdd {
    uint256 private balance = 1;

    function add(uint256 deposit) public {
        balance = balance + deposit;
    }
}

Source: https://smartcontractsecurity.github.io/SWC-registry/docs/SWC-101
pragma solidity ^0.4.24;

contract Overflow_Add {
    uint256 private Balance = 1;

    function AddSafe(uint256 deposit) public {
        uint256 newBalance = balance + deposit;
        require(newBalance >= deposit, "OVERFLOW DETECTED");

        balance += deposit;
    }
}

Source: https://smartcontractsecurity.github.io/SWC-registry/docs/SWC-101
pragma solidity ^0.4.5;

contract MegaTokenBank{
    mapping(address => uint256) public Ledger;
    uint256 constant PRICE_PER_TOKEN = 10000;

    function MegaTokenBank(address _player) public payable {
        require(msg.value == 1);
    }

    function buy(uint256 numTokens) public payable {
        require(msg.value == numTokens * PRICE_PER_TOKEN);
        Ledger[msg.sender] += numTokens;
    }

    function sell(uint256 numTokens) public {
        require(balanceOf[msg.sender] >= numTokens);
        Ledger[msg.sender] -= numTokens;
        msg.sender.transfer(numTokens * PRICE_PER_TOKEN);
    }
}

Problem:
Arithmetic Results in Integer Overflow

Solution
Ensure sanity checks are applied after arithmetic

Consider a library like SafeMath
(Source: https://github.com/OpenZeppelin/openzeppelin-solidity/tree/master/contracts/math)

Source: https://smartcontractsecurity.github.io/SWC-registry/docs/SWC-101
Exposed Private Data
There are no secrets on the blockchain

```
pragma solidity ^0.4.5;

contract SecretHolder {
    uint256 constant MySecretValue = 0xABCDEF1010;

    function GetSecret() public payable {
        require(msg.sender = owner);
    }
}
```

**Problem:**
The *World State* is stored in each synced node.

Hence your secret value is available by manual inspection.
Unprotected Self-Destruct (SWC-106)

```solidity
contract SuicideMultiTxFeasible {
    uint256 private initialized = 0;
    uint256 public count = 1;

    function init() public {
        initialized = 1;
    }

    function run(uint256 input) {
        if (initialized == 0) {
            return;
        }

        selfdestruct(msg.sender);
    }
}
```

**Problem:**
The self-destruct will destroy the contract and freeze any ether attached to the contract address.

**Whether it’s $1 dollar or $150 Million dollars**
Unprotected Self-Destruct (SWC-106) - Parity

“anyone can kill your contract #6995” – devops199
https://github.com/paritytech/parity-ethereum/issues/6995

Roughly 150-300 Million remains “Frozen”

Source: https://etherscan.io/address/0x863df6bfa4469f3ead0be8f9f2aae51c91a907b4#code
Bad Randomness (SWC-120)
On the blockchain nothing is truly random

/*
 * @source: https://capturetheether.com/challenges/lotteries/guess-the-random-number/
 * @author: Steve Marx
 */

pragma solidity ^0.4.21;

contract GuessTheRandomNumberChallenge {
    uint8 answer;

    function GuessTheRandomNumberChallenge() public payable {
        require(msg.value == 1 ether);
        answer = uint8(keccak256(block.blockhash(block.number - 1), now));
    }

    function isComplete() public view returns (bool) {
        return address(this).balance == 0;
    }

    function guess(uint8 n) public payable {
        require(msg.value == 1 ether);

        if (n == answer) {
            msg.sender.transfer(2 ether);
        }
    }
}

Problems:
Miners can manipulate block numbers.
PC are far faster than Ethereum and can “run ahead” of the block chain.

Source: https://smartcontractsecurity.github.io/SWC-registry/docs/SWC-120
Bad Randomness (SWC-120)
On the blockchain nothing is truly random

Solution:
Only generate the “random” number AFTER the guesses are committed.
This call RANDAO or Commit Pattern.

Source: https://github.com/randao/randao

Source: https://smartcontractsecurity.github.io/SWC-registry/docs/SWC-120
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)()); // Calls Sender Code
            credit[msg.sender]-=amount;
        }
    }

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

Problem:
Ether is sent via call on the senders amount() function before it is actually deducted of the balance.

Withdraw can be called over and over again in amount() before the amount is deducted.
Re-Entrancy (SWC-107)

function withdraw(uint amount) public{
    if (credit[msg.sender]>= amount) {
        credit[msg.sender]-=amount; // Update Balance First
        require(msg.sender.call.value(amount)()); // Calls Sender
    }
}

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

Solution:
Update value before calling sender contracts code.

Ideally use send() or transfer() as opposed to calling the sender's code.

Source: https://smartcontractsecurity.github.io/SWC-registry/docs/SWC-107
The DAO Hack – Re-Entrancy

- Abused “split” function of DAO contract
- $3.6 million Ether stolen
- $420 million to date
- Due to the way the contract was structured a 27 day hold was in place
- Community majority (89%) voted to “Hard Fork” (creating the divide between Ether and Ether Classic)
- Actors who stole the ether were actively involved in trying to influence the community to not hard fork

Source: https://etherscan.io
Unsafe Authorization (SWC-115)

contract MyContract {

    address owner;

    function MyContract() public {
        owner = msg.sender; // Properly set in constructor
    }

    function sendTo(address receiver, uint amount) public
    {
        require(tx.origin == owner); // Improper Check
        receiver.transfer(amount);
    }
}

Problem:
A crafted blockheader with chosen tx.origin may be mined
If the block is “mined” a an actor may take over the contract then.

Source: https://smartcontractsecurity.github.io/SWC-registry/docs/SWC-115
Unsafe Authorization (SWC-115)

contract MyContract {
    address owner;

    function MyContract() public {
        owner = msg.sender; // Properly set in constructor
    }

    function sendTo(address receiver, uint amount) public {
        require(msg.sender == owner); // Improper Check
        receiver.transfer(amount);
    }
}

Solution:
Use msg.sender to validate who sent the message

Source: https://smartcontractsecurity.github.io/SWC-registry/docs/SWC-115
Unsafe Contract Constructors (SWC-118)

/*
 * @source: https://github.com/trailofbits/not-so-smart-contracts/blob/master/wrong_constructor_name/incorrect_constructor.sol
 * @author: Ben Perez
 * Modified by Gerhard Wagner
 */

pragma solidity 0.4.24;

contract Missing{
    address private owner;

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

    function missing() public {
        owner = msg.sender;
    }

    function () payable {}

    function withdraw() public onlyowner {
        owner.transfer(this.balance);
    }
}

Problem:
By mis-spelling the constructor name a default constructor is auto-generated without the expected checks.

Source: https://smartcontractsecurity.github.io/SWC-registry/docs/SWC-118
Unsafe Contract Constructors (SWC-118)

function withdraw()
    public
    onlyowner
    {  
        owner.transfer(this.balance);
    }
}
Out-Of-Bounds Write-Anywhere (SWC-124)

Problem:
Without appropriate bounds check index offsets called directly or arrays will write into nearby storage.

Often this includes over-writing the owner variable potentially changing the owner of the contract or modify other information on the stack.

Will Smart Contract Control Flow Exploitation become a thing? (We haven’t seen the first buffer overflow yet).

Source: https://smartcontractsecurity.github.io/SWC-registry/docs/SWC-124

function UpdateLedgerAtIndex(uint idx, uint entry) public {
    Ledger[idx] = entry;
}
Out-Of-Bounds Write-Anywhere (SWC-124)

```solidity
def function UpdateLedgerAtIndex(uint idx, uint entry) public {
    require(idx < Ledger.length);
    Ledger[idx] = entry;
}
```

**Solution:**
Ensure adequate bounds checking

*Source: https://smartcontractsecurity.github.io/SWC-registry/docs/SWC-124*
And of course, there’s exchange hacks!
Honey Pots

Contracts that appear vulnerable but are not
• Just have to send a little bit of Ether in... 😊
• Use of anti-disassembly tricks to hinder analysis

Great talk on research to detect such contracts
• Smart Contracts honeypots for profit (and probably fun) - Ben Schimdt
• Source: https://www.youtube.com/watch?v=Lj0J7_a1AVQ
Security Tools

IDE
• Remix (online IDE) - https://remix.ethereum.org/

Smart Contract Static Analysis
• Slither - https://github.com/trailofbits/slither

Smart Contract Dynamic Analysis (Symbolic Execution)
• Mithril Classic - https://github.com/ConsenSys/mythril-classic
• Manticore - https://github.com/trailofbits/manticore

Smart Contract Dynamic Analysis (Fuzzing)
• Echidna - https://github.com/trailofbits/echidna
To The Future

• Smart Contract development is still very new
• Increased use of design patterns in Smart Contract development to address challenges like upgrading
• Educate developers on types of weaknesses
• Better tooling
• Use of standards when implementing Tokens (ERC* series tokens)
References

1) Smart Contract Weakness Classification
https://smartcontractsecurity.github.io/SWC-registry/

2) Trail Of Bits – Not So Smart Contracts
https://github.com/trailofbits/not-so-smart-contracts

3) Smashing Ethereum Smart Contracts for Fun and ACTUAL Profit
https://github.com/b-mueller/smashing-smart-contracts

4) Smart Contract Best Practices

5) Ethereum Yellow and Beige Papers
Beige Paper - https://github.com/chronaeon/beigepaper
Challenges!

1) Capture The Ether (By Steve Marx @smarx)
   https://capturetheether.com/challenges/

2) Security Innovation Blockchain CTF (By Security Innovation)
   https://blockchain-ctf.securityinnovation.com/

3) EtherNaut CTF (@ZeppelinOrg)
   https://ethernaut.zeppelin.solutions/
Thank you!

• Thank you to Judy (@daarkprincess) for bringing the cookies!
• Thank you to OWASP Toronto and George Brown for hosting!
• Thank you to everyone for attending!
Questions?

I’m listening...