BLOCKCHAIN

SMART CONTRACT SECURİTY AND AUDİTS

Dr. Enis Karaaslan

Mugla Sitki Kocman University

Rüzgar Üren

1107090006

ruzgaruren@posta.mu.edu.tr

01 HAZ 2021

MUGLA SITKI KOCMAN UNIVERSTY

Computer Engineering

CONTENTS

BLOCKCHAIN

Defining Blockchain Technology

Blockchain is a cryptocurrency block.

Blockchain is a cryptocurrency blockchain, explorer service, as well as a cryptocurrency wallet and a cryptocurrency Exchange supporting bitcoin , bitcoin cash , and Etherium. They also provide Bitcoin data charts, stats, and market information.

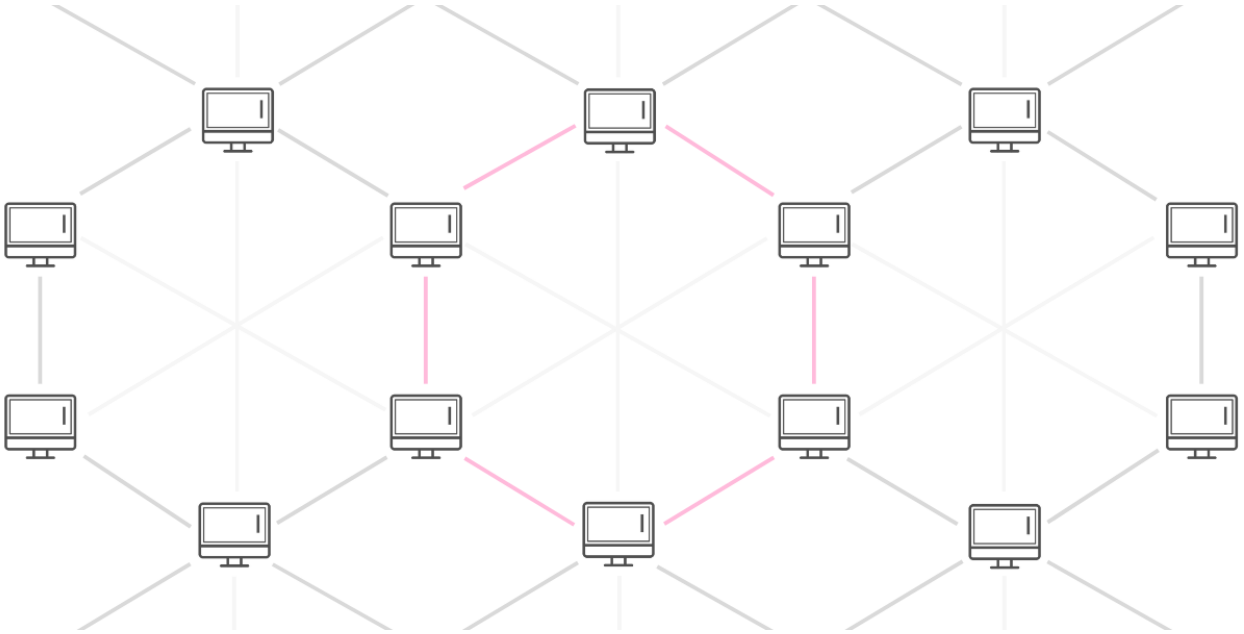
Blockchain technology is a type of distributed ledger technology (DLT) — It is an accounting system where the ledger (record of transactions) is distributed among a network of computers.

‘*Block’*

Each block records a number of transactions, similar to a page in a record-keeping book. The amount of transactions on a block varies from blockchain to blockchain. For example, each block on the Bitcoin blockchain holds up to 1 megabyte of information.

‘*Chain’*

These transactions are recorded in the form of hashes — strings of numbers and letters. The hash of each transaction is generated to include information from the current and past transactions. This creates a chain effect where the order of hashes cannot be changed. As a result, transactions are immutable once they’ve been added.



blockchain technology is a record-keeping tool.

blockchain networks are often referred to as peer-to-peer networks.

Steps for blockchain

Step 1: These computers verify all transactions one by one and add them onto a ‘block’ of information.

Step 2: T hese blocks are added to the blockchain

Step 3: Downloaded onto each computer

Lastly; In a nutshell, this is how these computers keep the blockchain secure and running.

What is nutshell?

Nutshell is a web and mobile customer relationship management (CRM) service. It is composed of a web application, as well as mobile applications for the iOS and Android platforms.

What makes blockchain technology unique?

Decentralized: blockchains are managed by a network of nodes rather than a central authority, they are fully decentralized

Transparent: Transactions on a blockchain are constantly being recorded and stored on the blockchain across nodes

Immutable: Blockchains are designed to enable permanent record keeping so that stored data cannot be altered after being added

Secure: It is hard to change or destroy blockchains because of its distributed nature.

Smart Contracts

A smart contract is a computer code that can be built into the blockchain to facilitate, verify, or negotiate a contract agreement. Smart contracts operate under a set of conditions that users agree to. When those conditions are met, the terms of the agreement are automatically carried out.

Smart contracts can also be called a programmable protocol.

Smart contracts provide what we call autonomous.

Thanks to the smart contracts we use to set the terms when trading cryptocurrencies, all your transactions are done automatically.

An example for s mart Contract:

Say, for example, a potential tenant would like to lease an apartment using a smart contract. The landlord agrees to give the tenant the door code to the apartment as soon as the tenant pays the security deposit. Both the tenant and the landlord would send their respective portions of the deal to the smart contract, which would hold onto and automatically exchange the door code for the security deposit on the date the lease begins. If the landlord doesn't supply the door code by the lease date, the smart contract refunds the security deposit. This would eliminate the fees and processes typically associated with the use of a notary, third-party mediator, or attornies.

Applicable Areas of Smart Contracts

The concept of smart contracts has started to be used widely, especially with ethereum, one of the most popular cryptocurrencies. Smart contracts have become a prominent feature of ethereum.

Since smart contracts were developed with cryptocurrencies, they are currently mostly used in the world of finance and banking. However, this technology can be used in as many fields as it allows around the world. For example, supply chains can use this structure to both track their products and automate all their transactions and payments. Likewise, real estate, healthcare, tax, insurance and countless other industries could well benefit from the use and benefits of smart contracts.

Developer and Users of Smart Contracts

Smart contracts were first announced in 1996 by computer scientist and cryptographer Nick Szabo. For several years Szabo reconsidered the concept and published several publications; defined business practices related to contract law through the design concept. However, the implementation of smart contracts did not occur until 2009; Bitcoin, the first cryptocurrency, eventually appeared with Blockchain, which provided a suitable environment for smart contracts. Interestingly, Nick Szabo devised a mechanism for a decentralized digital currency called Bit Gold in 1998. This was never materialized, but Bitcoin had a lot of features to brag about 10 years from now.

Smart contracts work on the 'If-Then' principle. That is, the ownership of the house is transferred to the buyer only when the agreed amount of money is sent to the system.

They also work as escrow, meaning both money and property rights are stored in the system and distributed to the participating parties at the same time.

Also, error-free delivery is guaranteed, as the transaction has been witnessed and verified by hundreds of people. There is no need for an intermediary as trust between the parties is no longer an issue. All the functions a real estate agent does can be pre-programmed into a smart contract, saving both the seller and the buyer a significant amount of money at the same time.

How Smart Contracts Work

Solidarity

A smart contract can work on its own, but it can also work with other smart contracts.

They can be set up in such a way that they are dependent on each other.

For example, the successful completion of a particular smart contract can trigger the launch of another

Objects of Smart Contracts

Every smart contract has three integral parts, also called objects. The first is those who sign two or more parties using the smart contract, who agree or disagree with the terms of the agreement using digital signatures.

A smart contract must contain certain terms.

These terms need to be fully defined mathematically and a programming language suitable for the specific smart contract environment must be used.

This includes the terms expected from all participating parties and any rules, rewards and penalties associated with such terms.

Environment

First of all, the environment needs to support the use of public key encryption, which enables them to log in for transaction using unique, specially generated encryption codes.

This is the definitive system used by the absolute majority of existing cryptocurrencies.

Second, it requires an open and decentralized database that all parties to the contract can fully trust and are fully automated.

Moreover, the entire environment needs to be decentralized for the smart contract to be implemented. Blockchain, especially the Ethereum Blockchain, are excellent environments for smart contracts.

What Do Smart Contracts Provide?

Autonomy: Smart contracts eliminate the need for a third-party facilitator mediation and basically give you full control of the deal.

Trust: No one can steal or lose your documents as they are stored in an encrypted and securely shared ledger. What's more, you don't have to trust or expect to trust the people you're dealing with, as the neutral system of smart contracts replaces trust.

Savings: Thanks to smart contracts, there is no need for notaries, real estate agents, consultants, help and many other middlemen.

Security: When implemented correctly, smart contracts are very difficult to interrupt. Also, the perfect environments for smart contracts are protected with sophisticated encryption that will keep your documents safe.

Efficiency: You will save a lot of time with smart contracts. Normally, you'd have to manually process stacks of paper documents and send them to and from specific locations, but with smart contracts these won't be necessary.

Areas Where Smart Contracts Can Benefit in Practice

Supply Chain Applications: If Smart Contracts are used in these applications, it will be possible to provide an automatization as well as transparency and security.

Internet of Things: Currently, studies are carried out on research topics such as blockchain-based smart home, smart city and smart transportation. In the case of the use of smart contracts in this area, IoT technologies will become more effective, more autonomous and more automatic.

Health Applications: Blockchain technology is beneficial in terms of patient privacy, distributed storage of information and protection of personal data.

Insurance Applications: traditional insurance solutions have uncertainties and long turnaround times. If the integration of smart contracts is ensured by eliminating the uncertainties in the processes, very fast and transparent results can be obtained without the intervention of a third party.

Finance Applications: Due to the nature of smart contracts, they can be used in many financial areas such as checks, loans, and leases.

Real Estate Applications: traditional real estate systems involve time consuming and risky processes. In addition, there are obligations such as paper waste and wet signature due to legal requirements. Thanks to smart contracts, trading can be made without the need for a third-party institution intermediary, and transactions can be transparently stored in digital ledgers.

Copyright Practices: Products containing copyright may require payment terms at different rates. The use of smart contracts can be beneficial in such applications.

Disadvantages

The code that creates the contract must be flawless and free of errors. Such errors can be exploited by scammers.

As with the DAO hack, money put into a smart account can be stolen by an error in the code of a smart contract.

Ex. For Dao attack:

```
contract ReentrancyVulnerability {

function withdraw () {

uint transferAmount = 10 ether;

if (!msg.sender.call.value(transferAmount)()) throw;

}

function deposit() payable {} // make contract payable and send ether

}
```

Above, the line highlighted in red is an external call, which should generally be avoided. The function withdraw() transfers 10 ether to msg.sender. So far so good. However, the receiver may call the function multiple times with a recursive send exploit, as seen below.

The innovation of technology brings with it many question marks. How will the government decide to issue such contracts? How will they be taxed? What if the contract does not reach the subject of the contract or if something unexpected happens?

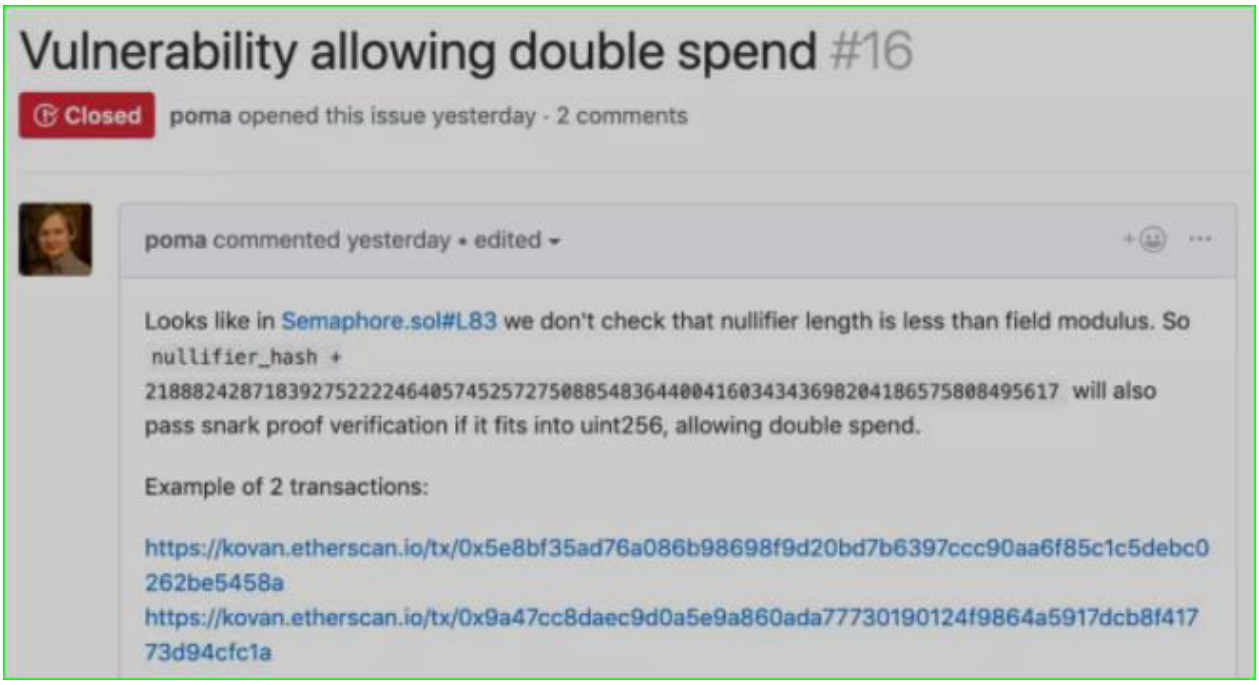
We only have so many questions.

The double-spend vulnerability: the first issue to arise :

Vulnerability allowing double spend #16

Problem 1:

Semaphore is an anonymous signaling system that uses "zero knowledge proof technology." The project evolved from the previous coin mixing project by renowned developer barryWhiteHat. Russian developer Poma first pointed out that the project may have a double-spending vulnerability.



Vulnerability allowing double spend #16

Closed poma opened this issue yesterday · 2 comments

poma commented yesterday • edited

Looks like in Semaphore.sol#L83 we don't check that nullifier length is less than field modulus. So `nullifier_hash + 21888242871839275222246405745257275088548364400416034343698204186575808495617` will also pass snark proof verification if it fits into uint256, allowing double spend.

Example of 2 transactions:

https://kovan.etherscan.io/tx/0x5e8bf35ad76a086b98698f9d20bd7b6397ccc90aa6f85c1c5debc0262be5458a
https://kovan.etherscan.io/tx/0x9a47cc8daec9d0a5e9a860ada77730190124f9864a5917dcb8f41773d94cfc1a

Problem 2:

Let's pay attention to the code written in line 83 of the codes below.

```
70      function broadcastSignal(
71          bytes memory signal,
72          uint[2] a,
73          uint[2][2] b,
74          uint[2] c,
75          uint[5] input // (root, nullifiers_hash, signal_hash, external_nullifier,
76      ) public {
77          uint256 start_gas = gasleft();
78
79          uint256 signal_hash = uint256(sha256(signal)) >> 8;
80          require(signal_hash == input[2]);
81          require(external_nullifier == input[3]);
82          require(verifyProof(a, b, c, input));
83          require(nullifiers_set[input[1]] == false);
84          address broadcaster = address(input[4]);
85          require(broadcaster == msg.sender);
86
87          bool found_root = false;
88          for (uint8 i = 0; i < root_history_size; i++) {
89              if (root_history[i] == input[0]) {
90                  found_root = true;
91                  break;
92              }
93          }
94          require(found_root);
95
96          insert(signal_tree_index, signal_hash);
97          nullifiers_set[input[1]] = true;
```


This function requires the caller to generate zero-knowledge proof to prove he can withdraw money from the contract. To avoid the occurrence of "double spend", the function also reads the "discard list" and checks whether a particular element of the certificate has been marked. If the evidence is on the abandonment list, the contract determines that the verification failed and the caller cannot withdraw the money.

The developer believes that in this way the same proof cannot be presented over and over for profit, and this action can effectively prevent double spend or replay attacks. However, things backfire and a deadly problem is being overlooked here.

The attacker can exploit the "input alias" vulnerability based on the evidence submitted successfully, and make a small change to the original input to quickly "make the proof", successfully pass the zero-knowledge proof verification on line 82 of the contract, and bypass the line 83 double-spend prevention check.

This issue can be followed until 2017. Example of zkSNARKs contract encryption provided by Christian Reitwiessner, the inventor of the Solidity language I

Since then, almost all contracts on Ethereum using zkSNARKs technology have adopted this practice.

Last Words:

The technology will certainly be perfected in time. Without a doubt, smart contracts are about to become an integral part of our society.

Bitcoin review with Rstudio

```
library(readr)

bitcoin <- read_csv("C:/Users/winds/Desktop/SON TESLİM TARİHLERİ/CaNS 2 HAZIRAN/veriler incelenecek/bitcoin.csv")

View(bitcoin)


head(bitcoin,5)


str(bitcoin)


nrow(bitcoin)

ncol(bitcoin)


bit1 <-bitcoin[c(bitcoin$generatedCoin,bitcoin$txCount)]



head(bitcoin[bitcoin$txCount>"5", ], 10)


bit2<-(bitcoin$generatedCoins )

bit2

bit3<-(bitcoin$txCount )

bit3

ggplot(data =bitcoin, aes(generatedCoins,txCount) )


par(mfrow = c(2,2))

plot(bitcoin$generatedCoins)
```

```
plot(bitcoin$generatedCoins,bitcoin$txCount)
plot(bitcoin$generatedCoins,bitcoin$txCount)
cor(bitcoin$generatedCoins,bitcoin$txCount) # coralation



par(mfrow = c(1,1))

plot(bitcoin$generatedCoins[bitcoin$txCount > "150"])

plot(bitcoin$txCount[bitcoin$generatedCoins > "5000"])

mean(bitcoin$blockSize)
mean(bitcoin$averageDifficulty)




#Single-variable plots Let’s continue with the bitcoin data
#Here are some basic single-variable plots.

par(mfrow = c(2,4)) #
plot(bitcoin$fees)
with(bitcoin, hist(activeAddresses))
with(bitcoin, boxplot(generatedCoins))
plot(bitcoin$date)
plot(bitcoin$txtCount)
plot(bitcoin$blockSize,bitcoin$generatedCoins)
qqnorm(bitcoin$blockCount)
```

```
qqline(bitcoin$generatedCoins)
qqnorm(bitcoin$blockCount)
qqline(bitcoin$generatedCoins)


#Here’s the qplot call that does it all in one simple line

qplot(x=blockCount, y=blockSize, data=bitcoin,
      color = fees,
      xlab = "blockCount ",
      ylab = "blockSize"
)




ggplot(bitcoin, aes(x=blockCount, y=blockSize, color=fees)) +
  geom_point() + # Adds points (scatterplot)
  geom_smooth(method = "lm") + # Adds regression lines
  ylab("blockCount") + #
  xlab("blockSize") + #
  ggtitle("Blockcount with block size ")
```

Example fors mart Contract:

1-pragma solidity ^0.5.0;

```
import "./IERC20.sol";
```

```
import "./SafeMath.sol";

/**
 * @dev Implementation of the {IERC20} interface.
 *
 * This implementation is agnostic to the way tokens are created. This means
 * that a supply mechanism has to be added in a derived contract using {_mint}.
 * For a generic mechanism see {ERC20Mintable}.
 *
 * TIP: For a detailed writeup see our guide
 * https://forum.zeppelin.solutions/t/how-to-implement-erc20-supply-mechanisms/226[How
 * to implement supply mechanisms].
 *
 * We have followed general OpenZeppelin guidelines: functions revert instead
 * of returning `false` on failure. This behavior is nonetheless conventional
 * and does not conflict with the expectations of ERC20 applications.
 *
 * Additionally, an {Approval} event is emitted on calls to {transferFrom}.
 * This allows applications to reconstruct the allowance for all accounts just
 * by listening to said events. Other implementations of the EIP may not emit
 * these events, as it isn't required by the specification.
 *
 * Finally, the non-standard {decreaseAllowance} and {increaseAllowance}
 * functions have been added to mitigate the well-known issues around setting
 * allowances. See {IERC20-approve}.
 */
contract ERC20 is IERC20 {
    using SafeMath for uint256;
```

```
mapping (address => uint256) private _balances;

mapping (address => mapping (address => uint256)) private _allowances;

uint256 private _totalSupply;

/**
 * @dev See {IERC20-totalSupply}.
 */
function totalSupply() public view returns (uint256) {
   return _totalSupply;
}

/**
 * @dev See {IERC20-balanceOf}.
 */
function balanceOf(address account) public view returns (uint256) {
   return _balances[account];
}

/**
 * @dev See {IERC20-transfer}.
 *
 * Requirements:
 *
 * - `recipient` cannot be the zero address.
 * - the caller must have a balance of at least `amount`.
 */
function transfer(address recipient, uint256 amount) public returns (bool) {
```

```
    _transfer(msg.sender, recipient, amount);
    return true;
}

/**
 * @dev See {IERC20-allowance}.
 */
function allowance(address owner, address spender) public view returns (uint256) {
    return _allowances[owner][spender];
}

/**
 * @dev See {IERC20-approve}.
 *
 * Requirements:
 *
 * - `spender` cannot be the zero address.
 */
function approve(address spender, uint256 value) public returns (bool) {
    _approve(msg.sender, spender, value);
    return true;
}

/**
 * @dev See {IERC20-transferFrom}.
 *
 * Emits an {Approval} event indicating the updated allowance. This is not
 * required by the EIP. See the note at the beginning of {ERC20};
 *
```

```
 * Requirements:
 * - `sender` and `recipient` cannot be the zero address.
 * - `sender` must have a balance of at least `value`.
 * - the caller must have allowance for `sender`'s tokens of at least
 * `amount`.
 */
function transferFrom(address sender, address recipient, uint256 amount) public returns (bool) {
    _transfer(sender, recipient, amount);
    _approve(sender, msg.sender, _allowances[sender][msg.sender].sub(amount));
    return true;
}

/**
 * @dev Atomically increases the allowance granted to `spender` by the caller.
 *
 * This is an alternative to {approve} that can be used as a mitigation for
 * problems described in {IERC20-approve}.
 *
 * Emits an {Approval} event indicating the updated allowance.
 *
 * Requirements:
 *
 * - `spender` cannot be the zero address.
 */
function increaseAllowance(address spender, uint256 addedValue) public returns (bool) {
    _approve(msg.sender, spender, _allowances[msg.sender][spender].add(addedValue));
    return true;
}
```

```
/**
 * @dev Atomically decreases the allowance granted to `spender` by the caller.
 *
 * This is an alternative to {approve} that can be used as a mitigation for
 * problems described in {IERC20-approve}.
 *
 * Emits an {Approval} event indicating the updated allowance.
 *
 * Requirements:
 *
 * - `spender` cannot be the zero address.
 * - `spender` must have allowance for the caller of at least
 * `subtractedValue`.
 */
function decreaseAllowance(address spender, uint256 subtractedValue) public returns (bool) {
    _approve(msg.sender, spender, _allowances[msg.sender][spender].sub(subtractedValue));
    return true;
}

/**
 * @dev Moves tokens `amount` from `sender` to `recipient`.
 *
 * This is internal function is equivalent to {transfer}, and can be used to
 * e.g. implement automatic token fees, slashing mechanisms, etc.
 *
 * Emits a {Transfer} event.
 *
 * Requirements:
 *
```

```
     * - `sender` cannot be the zero address.
     * - `recipient` cannot be the zero address.
     * - `sender` must have a balance of at least `amount`.
     */
    function _transfer(address sender, address recipient, uint256 amount) internal {
        require(sender != address(0), "ERC20: transfer from the zero address");
        require(recipient != address(0), "ERC20: transfer to the zero address");

        _balances[sender] = _balances[sender].sub(amount);
        _balances[recipient] = _balances[recipient].add(amount);
        emit Transfer(sender, recipient, amount);
    }

    /** @dev Creates `amount` tokens and assigns them to `account`, increasing
     * the total supply.
     *
     * Emits a {Transfer} event with `from` set to the zero address.
     *
     * Requirements
     *
     * - `to` cannot be the zero address.
     */
    function _mint(address account, uint256 amount) internal {
        require(account != address(0), "ERC20: mint to the zero address");

        _totalSupply = _totalSupply.add(amount);
        _balances[account] = _balances[account].add(amount);
        emit Transfer(address(0), account, amount);
    }
```

```
/**
 * @dev Destroys `amount` tokens from `account`, reducing the
 * total supply.
 *
 * Emits a {Transfer} event with `to` set to the zero address.
 *
 * Requirements
 *
 * - `account` cannot be the zero address.
 * - `account` must have at least `amount` tokens.
 */
function _burn(address account, uint256 value) internal {
   require(account != address(0), "ERC20: burn from the zero address");


   _totalSupply = _totalSupply.sub(value);
   _balances[account] = _balances[account].sub(value);
   emit Transfer(account, address(0), value);
}


/**
 * @dev Sets `amount` as the allowance of `spender` over the `owner`s tokens.
 *
 * This is internal function is equivalent to `approve`, and can be used to
 * e.g. set automatic allowances for certain subsystems, etc.
 *
 * Emits an {Approval} event.
 *
 * Requirements:
```

```
     *
     * - `owner` cannot be the zero address.
     * - `spender` cannot be the zero address.
     */
    function _approve(address owner, address spender, uint256 value) internal {
        require(owner != address(0), "ERC20: approve from the zero address");
        require(spender != address(0), "ERC20: approve to the zero address");

        _allowances[owner][spender] = value;
        emit Approval(owner, spender, value);
    }

    /**
     * @dev Destoys `amount` tokens from `account`.`amount` is then deducted
     * from the caller's allowance.
     *
     * See {_burn} and {_approve}.
     */
    function _burnFrom(address account, uint256 amount) internal {
        _burn(account, amount);
        _approve(account, msg.sender, _allowances[account][msg.sender].sub(amount));
    }
}
```

2-pragma solidity ^0.5.0;

```
/**
 * @dev Wrappers over Solidity's arithmetic operations with added overflow
```

```
 * checks.
 *
 * Arithmetic operations in Solidity wrap on overflow. This can easily result
 * in bugs, because programmers usually assume that an overflow raises an
 * error, which is the standard behavior in high level programming languages.
 * `SafeMath` restores this intuition by reverting the transaction when an
 * operation overflows.
 *
 * Using this library instead of the unchecked operations eliminates an entire
 * class of bugs, so it's recommended to use it always.
 */
library SafeMath {
    /**
     * @dev Returns the addition of two unsigned integers, reverting on
     * overflow.
     *
     * Counterpart to Solidity's `+` operator.
     *
     * Requirements:
     * - Addition cannot overflow.
     */
    function add(uint256 a, uint256 b) internal pure returns (uint256) {
        uint256 c = a + b;
        require(c >= a, "SafeMath: addition overflow");

        return c;
    }
```

```
/**
 * @dev Returns the subtraction of two unsigned integers, reverting on
 * overflow (when the result is negative).
 *
 * Counterpart to Solidity's `-` operator.
 *
 * Requirements:
 * - Subtraction cannot overflow.
 */
function sub(uint256 a, uint256 b) internal pure returns (uint256) {
    require(b <= a, "SafeMath: subtraction overflow");
    uint256 c = a - b;

    return c;
}

/**
 * @dev Returns the multiplication of two unsigned integers, reverting on
 * overflow.
 *
 * Counterpart to Solidity's `*` operator.
 *
 * Requirements:
 * - Multiplication cannot overflow.
 */
function mul(uint256 a, uint256 b) internal pure returns (uint256) {
    // Gas optimization: this is cheaper than requiring 'a' not being zero, but the
    // benefit is lost if 'b' is also tested.
```

```
    // See: https://github.com/OpenZeppelin/openzeppelin-contracts/pull/522
    if (a == 0) {
        return 0;
    }

    uint256 c = a * b;
    require(c / a == b, "SafeMath: multiplication overflow");

    return c;
}

/**
 * @dev Returns the integer division of two unsigned integers. Reverts on
 * division by zero. The result is rounded towards zero.
 *
 * Counterpart to Solidity's `/` operator. Note: this function uses a
 * `revert` opcode (which leaves remaining gas untouched) while Solidity
 * uses an invalid opcode to revert (consuming all remaining gas).
 *
 * Requirements:
 * - The divisor cannot be zero.
 */
function div(uint256 a, uint256 b) internal pure returns (uint256) {
    // Solidity only automatically asserts when dividing by 0
    require(b > 0, "SafeMath: division by zero");
    uint256 c = a / b;
    // assert(a == b * c + a % b); // There is no case in which this doesn't hold
```

```
        return c;
    }

    /**
     * @dev Returns the remainder of dividing two unsigned integers. (unsigned integer modulo),
     * Reverts when dividing by zero.
     *
     * Counterpart to Solidity's `%` operator. This function uses a `revert`
     * opcode (which leaves remaining gas untouched) while Solidity uses an
     * invalid opcode to revert (consuming all remaining gas).
     *
     * Requirements:
     * - The divisor cannot be zero.
     */
    function mod(uint256 a, uint256 b) internal pure returns (uint256) {
        require(b != 0, "SafeMath: modulo by zero");
        return a % b;
    }
}
```

……..

3- pragma solidity ^0.5.0;

```
/**
```

```
 * @dev Wrappers over Solidity's arithmetic operations with added overflow
 * checks.
 *
 * Arithmetic operations in Solidity wrap on overflow. This can easily result
 * in bugs, because programmers usually assume that an overflow raises an
 * error, which is the standard behavior in high level programming languages.
 * `SafeMath` restores this intuition by reverting the transaction when an
 * operation overflows.
 *
 * Using this library instead of the unchecked operations eliminates an entire
 * class of bugs, so it's recommended to use it always.
 */
library SafeMath {
    /**
     * @dev Returns the addition of two unsigned integers, reverting on
     * overflow.
     *
     * Counterpart to Solidity's `+` operator.
     *
     * Requirements:
     * - Addition cannot overflow.
     */
    function add(uint256 a, uint256 b) internal pure returns (uint256) {
        uint256 c = a + b;
        require(c >= a, "SafeMath: addition overflow");

        return c;
    }
```

```
/**
 * @dev Returns the subtraction of two unsigned integers, reverting on
 * overflow (when the result is negative).
 *
 * Counterpart to Solidity's `-` operator.
 *
 * Requirements:
 * - Subtraction cannot overflow.
 */
function sub(uint256 a, uint256 b) internal pure returns (uint256) {
    require(b <= a, "SafeMath: subtraction overflow");
    uint256 c = a - b;

    return c;
}

/**
 * @dev Returns the multiplication of two unsigned integers, reverting on
 * overflow.
 *
 * Counterpart to Solidity's `*` operator.
 *
 * Requirements:
 * - Multiplication cannot overflow.
 */
function mul(uint256 a, uint256 b) internal pure returns (uint256) {
    // Gas optimization: this is cheaper than requiring 'a' not being zero, but the
```

```
    // benefit is lost if 'b' is also tested.
    // See: https://github.com/OpenZeppelin/openzeppelin-contracts/pull/522
    if (a == 0) {
        return 0;
    }

    uint256 c = a * b;
    require(c / a == b, "SafeMath: multiplication overflow");

    return c;
}

/**
 * @dev Returns the integer division of two unsigned integers. Reverts on
 * division by zero. The result is rounded towards zero.
 *
 * Counterpart to Solidity's `/` operator. Note: this function uses a
 * `revert` opcode (which leaves remaining gas untouched) while Solidity
 * uses an invalid opcode to revert (consuming all remaining gas).
 *
 * Requirements:
 * - The divisor cannot be zero.
 */
function div(uint256 a, uint256 b) internal pure returns (uint256) {
    // Solidity only automatically asserts when dividing by 0
    require(b > 0, "SafeMath: division by zero");
    uint256 c = a / b;
    // assert(a == b * c + a % b); // There is no case in which this doesn't hold
```

```
        return c;
    }

    /**
     * @dev Returns the remainder of dividing two unsigned integers. (unsigned integer modulo),
     * Reverts when dividing by zero.
     *
     * Counterpart to Solidity's `%` operator. This function uses a `revert`
     * opcode (which leaves remaining gas untouched) while Solidity uses an
     * invalid opcode to revert (consuming all remaining gas).
     *
     * Requirements:
     * - The divisor cannot be zero.
     */
    function mod(uint256 a, uint256 b) internal pure returns (uint256) {
        require(b != 0, "SafeMath: modulo by zero");
        return a % b;
    }
}
```

/4-/0.5.1-c8a2

// Enable optimization

```
pragma solidity ^0.5.0;


import "./ERC20.sol";
import "./ERC20Detailed.sol";


/**
 * @title SimpleToken
 * @dev Very simple ERC20 Token example, where all tokens are pre-assigned to the creator.
 * Note they can later distribute these tokens as they wish using `transfer` and other
 * `ERC20` functions.
 */
contract Token is ERC20, ERC20Detailed {


    /**
     * @dev Constructor that gives msg.sender all of existing tokens.
     */
    constructor () public ERC20Detailed("DARK CELL", "DC", 18) {
        _mint(msg.sender, 2500000000000000 * (10 ** uint256(decimals())));
    }
}
```

Video on youtube:

https://youtu.be/yaOK1mI2U9M

References :

http://wiki.netseclab.mu.edu.tr/index.php?title=MSK%C3%9C_Blok_Zinciri_Ara%C5%9Ft%C4%B1rma_Grubu

https://dergi.bmo.org.tr/teknoloji/pyhton-programlama-dili-neden-giderek-popularitesini-arttiriyor

https://en.wikipedia.org/wiki/Blockchain.com

https://en.wikipedia.org/wiki/Nutshell_CRM#:~:text=Nutshell%20is%20a%20web%20and,the%20iOS%20and%20Android%20platforms.

https://github.com/appliedzkp/semaphore/issues/16

http://ege-law.com/smart-contracts-akilli-sozlesmeler-uygulama-alanlari-avantaj-ve-dezavantajlari/

https://medium.com/@knownsec404team/ethereum-smart-contract-audit-checklist-ba9d1159b901

https://www2.deloitte.com/tr/tr/pages/finance/articles/cfo-insights-getting-smart-contracts.html#

https://www.devteam.space/blog/how-to-audit-a-smart-contract-a-guide/

https://medium.com/search?q=Smart%20Contract%20Security%20and%20Audits

https://blockgeeks.com/guides/audit-smart-contract/

https://youtu.be/yaOK1mI2U9M

Rüzgar Üren

ruzgaruren@posta.mu.edu.tr