COMMON ERC

20/9/2024

Speaker: Sham Chun Ning Adrian

01 - ERC20 (TOKEN)

02 - ERC721 (NFT)

03 - ERC4626 (VAULT)

04 - ERC3156 (FLASH LOAN INTERFACE)

What is EIP (and ERC)?

- EIP stands for “Ethereum Improvement Proposal”
  - Proposes improvements for the Ethereum ecosystem
- ERC is a subcategory of EIP
  - Mainly focuses on providing a standardization for token and smart contract design

Motivation

- Better understanding of how tokens work
- Better understanding of how to design smart contracts

https://eips.ethereum.org/all

ERC20 FUNGIBLE TOKEN STANDARD

ERC20 is a technical standard for establishing fungible assets on the Ethereum blockchain.

- Fungible means that every single ERC20 token is exactly the same as another ERC20 token from the same contract.
  - Fiat currencies representation (USDC or USDT - stablecoin)
  - Financial assets (like shares of a company)
- Standard helps create compatibility and safeness for developers developing ERC20 tokens

CODE OVERVIEW

Total consists of 9 functions and 2 events.

- 3 of which are state change functions

01 - TRANSFER

02 - APPROVE

03 - TRANSFERFROM

```
//----------------VIEW FUNCTIONS---------------
function name() public view returns (string);
function symbol() public view returns (string);
function decimals() public view returns (uint8);
function totalSupply() public view returns (uint256);
function allowance(address _owner, address _spender) public view returns (uint256 remaining);
function balanceOf(address _owner) public view returns (uint256 balance);

//-------------STATE CHANGE FUNCTIONS------------
function transfer(address _to, uint256 _value) public returns (bool success);
    //  _to cannot be zero address
    //  caller must have a balance of at least _value

function transferFrom(address _from, address _to, uint256 _value) public returns (bool success);
    //  _from and _to cannot be zero address
    //  _from must have a balance of at least _value
    //  the caller must have allowance for _from's tokens of at least _value

function approve(address _spender, uint256 _value) public returns (bool success);
    //  _spender cannot be zero address

//--------------------EVENTS--------------------
event Transfer(address indexed _from, address indexed _to, uint256 _value);
     //  emit when successfully called transfer or transferFrom
event Approval(address indexed _owner, address indexed _spender, uint256 _value);
     //  emit when successfully called approve
```

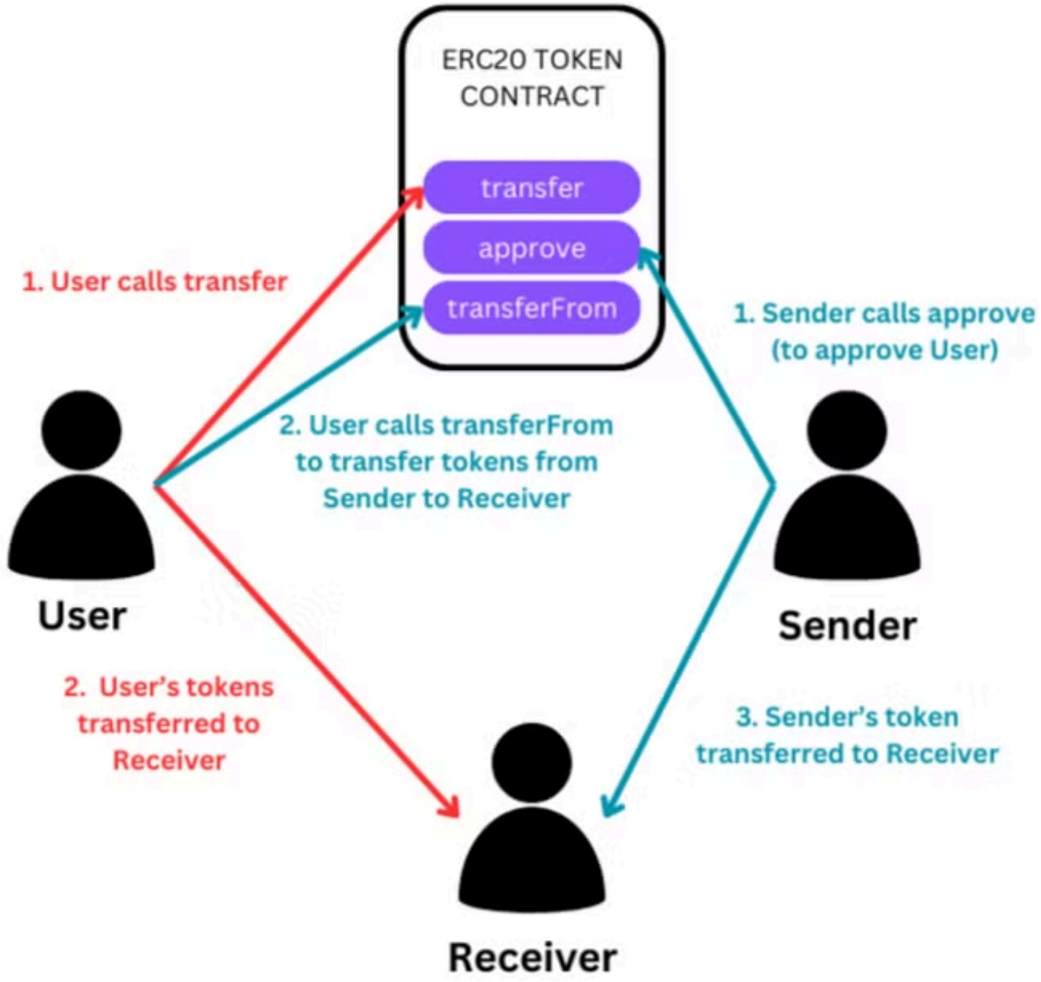
TRANSFER SCENARIOS

transfer (address _to, uint256 _value):
The caller (msg.sender) transfers *_value* amount of ERC20 token to the address *_to*

Example:

1. User owns 100 USDC
2. User calls *transfer(receiver, 100)* on the USDC smart contract
3. 100 USDC token is transferred directly to receiver from user

Roles:
Sender and Receiver. Sender is the address that calls *transfer()* and receiver is the address that receives the tokens.

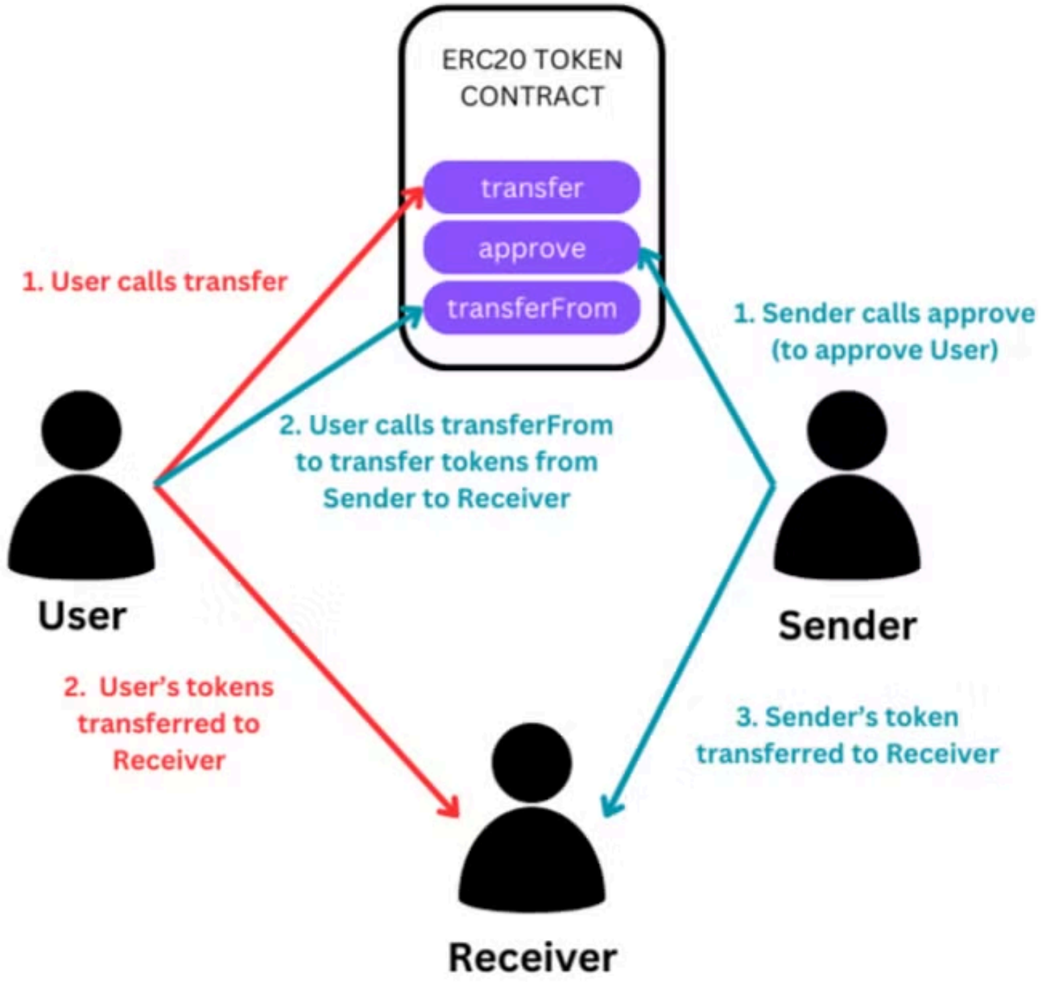


TRANSFER SCENARIOS

approve(address _spender, uint256 _value):
The caller approves the address *_spender* to use *_value* amount of ERC20 tokens from the caller's balance

transferFrom (address _from, address _to, uint256 _value):
The caller transfers *_value* amount of ERC20 token from the address *_from* to the address *_to*

Roles:
Sender Spender and Receiver. Sender approves user first. user becomes the spender. Spender is the address that calls *transferFrom()* to spend Sender's balance. Receiver is the address that receives the tokens.



BANK EXAMPLE

For an ETH bank smart contract, the deposit function can be set to payable and receive ETH.

- Caller can call deposit{value: xx}();

However, there is no direct way to send erc20 token during a function call. In order to allow the deposit function to be atomic, developers will use the transferFrom function to perform token deposit.

```
contract Bank {
    address tokenAddress; //assume tokenAddress set during constructor
    mapping (address => uint256) balances;

    //  We can use transferFrom to handle deposit actions
    function deposit(uint256 amount) public {
        IERC20(tokenAddress).transferFrom(msg.sender, address(this), amount);
        balance[msg.sender] += amount;
    }

    //  We can use transfer to handle withdrawal actions
    function withdraw(uint256 amount) public {
        require(balances[msg.sender] >= amount, "Insufficient Balance");
        IERC20(tokenAddress).transfer(msg.sender, amount);
        balance[msg.sender] -= amount;
    }
}
```

ERC721 NON-FUNGIBLE TOKEN STANDARD

ERC721 is a technical standard for establishing non-fungible assets on the Ethereum blockchain.

- Non-fungible means that every single ERC721 token is unique.
  - Artworks
  - Real world asset representation
  - Digital assets
  - Digital identity
- Standard helps create compatibility and safeness for developers developing ERC721 tokens

CODE OVERVIEW

Total consists of 9 functions and 3 events.

- 5 of which are state change functions

01 - APPROVE

02 - SETAPPROVALFORALL

03 - TRANSFERFROM

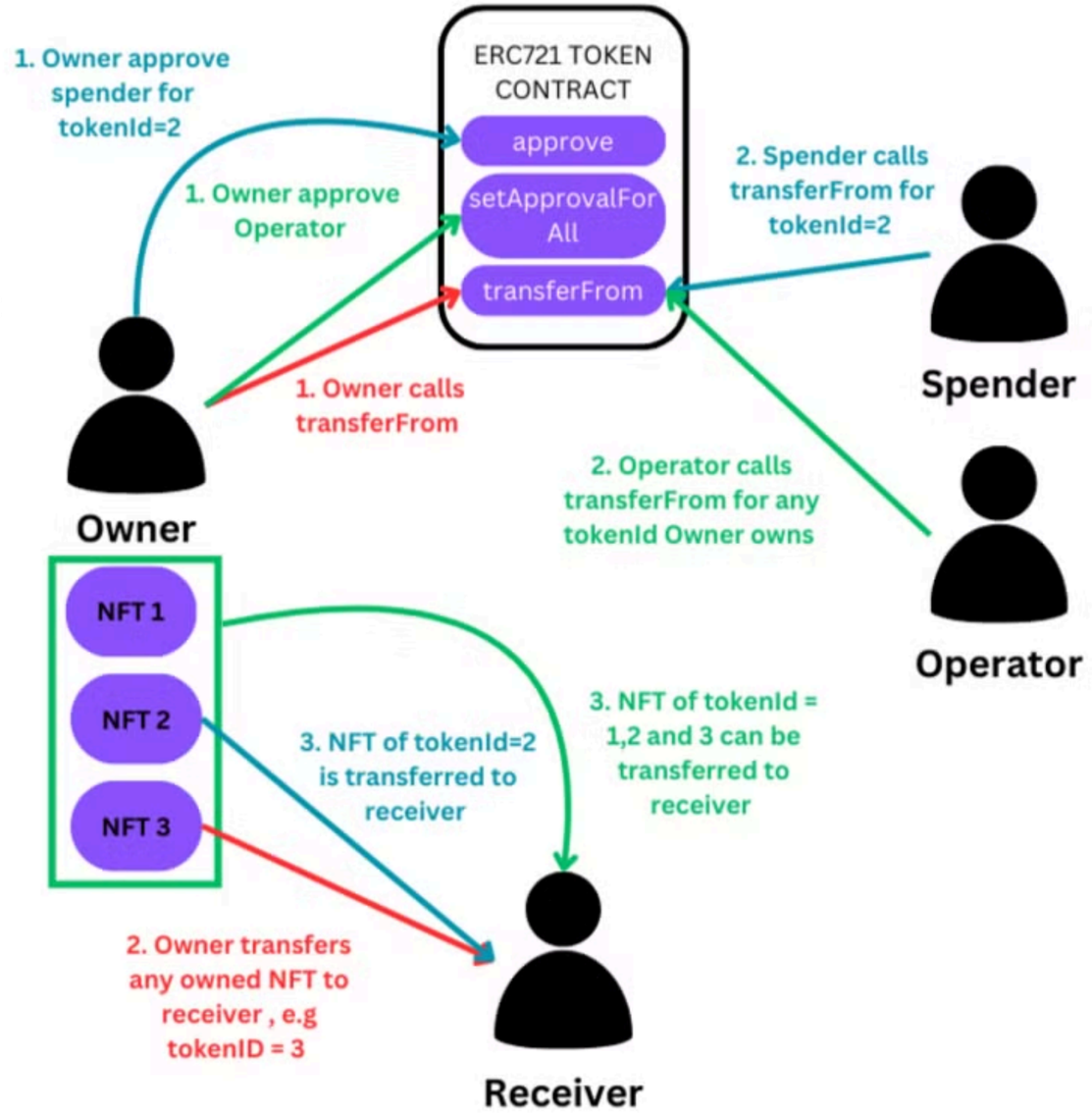
```solidity
//-----------------------------EVENT-------------------------------
event Transfer(address indexed _from, address indexed _to, uint256 indexed _tokenId);
//  This emits when ownership of any NFT changes by any mechanism.
//  Minted (`from` == 0) and burnt (`to` == 0).
//  Exception: during contract creation, any number of NFTs
//  may be created and assigned without emitting Transfer. At the time of
//  any transfer, the approved address for that NFT (if any) is reset to none.

event Approval(address indexed _owner, address indexed _approved, uint256 indexed _tokenId);
//  This emits when the approved address for an NFT is changed or
//  reaffirmed. The zero address indicates there is no approved address.
//  When a Transfer event emits, this also indicates that the approved
//  address for that NFT (if any) is reset to none.

event ApprovalForAll(address indexed _owner, address indexed _operator, bool _approved);
//  This emits when an operator is enabled or disabled for an owner.
//  The operator can manage all NFTs of the owner.

//-------------------------VIEW FUNCTIONS---------------------------
function balanceOf(address _owner) public view returns (uint256);
function ownerOf(uint256 _tokenId) public view returns (address);
function getApproved(uint256 _tokenId) public view returns (address);
function isApprovedForAll(address _owner, address _operator) public view returns (bool);

//----------------------STATE CHANGE FUNCTIONS-----------------------
function safeTransferFrom(address _from, address _to, uint256 _tokenId, bytes data) public;
        //  Requirements: _from and _to cannot be zero address and _tokenId
        //  Must be owned by _from
        //  1. Checks if caller is owner of NFT
        //  2. Checks if caller is approved spender of NFT
        //  3. Checks if caller is operator of owner of NFT
        //  4. Checks if _to is a contract (byte code size > 0). If so, it calls
    //  `onERC721Received` and reverts if the return value is not
    //  `bytes4(keccak256("onERC721Received(address,address,uint256,bytes)"))`.
function safeTransferFrom(address _from, address _to, uint256 _tokenId) public;
        //  same as the above but bytes = ""
function transferFrom(address _from, address _to, uint256 _tokenId) public;
        //  same as safeTransferFrom but do not check for 'onERC721Received'
function approve(address _approved, uint256 _tokenId) public;
function setApprovalForAll(address _operator, bool _approved) public;
```

TRANSFER SCENARIOS

- **approve(address _spender, uint256 _tokenId):**
  The caller (msg.sender) approves the address *_spender* as the spender of the *_tokenId* NFT

- **setApprovalForAll(address _operator, bool _approved):**
  The caller (msg.sender) approves or remove (depending on _approved) the address *_operator* as the operator of the caller. An operator can spend any NFT owned by the caller.

- **transferFrom(address _from, address _to, uint256 _tokenId):**
  Caller must be owner, spender or operator. Caller transfers the *_tokenId* NFT from the address *_from* to the address *_to*.



TRANSFERFROM FUNCTION

Three Parameters:
From
To
Token Id

- transferFrom() will perform the following checkings
  - Check if caller(from) is Owner, Spender or Operator

- **What is the difference between safeTransferFrom()? and transferFrom()?**
  - safeTransferFrom will perform an additional checking:
    i. Check if receiver is a contract
    ii. if yes, calls the contract ‘onERC721Received’ function

COMPARISON

ERC20	ERC721
Fungible	Non-Fungible
Has a transfer() and transferFrom() function	Has only transferFrom() function
2 roles can call transfer() or transferFrom(): Owner and Spender	3 roles can call transferFrom(): Owner, Spender and Operator
Can approve as many addresses	Can only approve one address per tokenID at a time. An owner can approve as many operators at a time

ERC4626 VAULT STANDARD

ERC4626 is a technical standard for establishing vault smart contracts on the Ethereum blockchain.

- Standard designed for yield-bearing vaults
  - For tokenized vaults that represent shares of a single underlying ERC-20 token
  - Somewhat similar to buying shares of a company, usually used in investing in shares of a protocol or service.

- Learn about “inflation attack”

CODE OVERVIEW

01 - CONVERTTOSHARES

02 - CONVERTTOASSETS

MUST emit event

MUST include fee

MUST NOT revert

MUST NOT include fee

MUST round down towards 0

User-specific scope

- maxDeposit()
- deposit()
- maxMint()
- mint()
- maxWithdraw()
- withdraw()
- maxRedeem()
- redeem()

Global scope

- asset()
- totalAssets()
- convertToShares()
- convertToAssets()
- previewDeposit()
- previewMint()
- previewWithdraw()
- previewRedeem()

INFLATION ATTACK

Attack scenario:

1. A attacker back-runs the transaction of an ERC4626 pool creation.
2. The attacker mints for themselves one share: deposit(1). Thus, totalAsset()==1, totalSupply()==1.
3. The attacker front-runs the deposit of the victim who wants to deposit 20,000 USDT (20,000.000000).
4. The attacker inflates the denominator: asset.transfer(20_000e6). Now totalAsset()==20_000e6 + 1, totalSupply()==1.
5. Next, the victim's tx takes place. The victim gets 1 * 20_000e6 / (20_000e6 + 1) == 0 shares. The victim gets zero shares.
6. The attacker burns their share and gets all the money.

https://mixbytes.io/blog/overview-of-the-inflation-attack

```solidity
abstract contract ERC4626 is ERC20 {
    IERC20 asset;

    constructor(IERC20 asset_) {
        asset = asset_;
    }

    function totalAssets() public view returns (uint256) {
        return asset.balanceOf(address(this));
    }

    function convertToShares(uint256 assets) public view returns (uint256) {
        if (totalAssets() == 0) {
            return assets;
        }
        return totalSupply() * assets / totalAssets();
    }

    function convertToAssets(uint256 shares) public view returns (uint256) {
        return totalAssets() * shares / totalSupply();
    }

    function deposit(uint256 assets) public {
        uint256 shares = convertToShares(assets);
        asset.transferFrom(msg.sender, address(this), assets);
        _mint(msg.sender, shares);
    }

    function burn(uint256 shares) public {
        asset.transfer(msg.sender, convertToAssets(shares));
    }
}
```

INFLATION ATTACK

Attack scenario:

1. A attacker back-runs the transaction of an ERC4626 pool creation.
2. The attacker mints for themselves one share: deposit(1). Thus, totalAsset()==1, totalSupply()==1.
3. The attacker front-runs the deposit of the victim who wants to deposit 20,000 USDT (20,000.000000).
4. The attacker inflates the denominator: asset.transfer(20_000e6). Now totalAsset()==20_000e6 + 1, totalSupply()==1.
5. Next, the victim's tx takes place. The victim gets 1 * 20_000e6 / (20_000e6 + 1) == 0 shares. The victim gets zero shares.
6. The attacker burns their share and gets all the money.

	Assets	Shares
Initial State	0	0
After Attacker Deposit	1	1
Victim Wants to Deposit 20000 USDT	Expected Assets = 20000e6 + 1	Expected Shares = 20000e6/1 = 20000e6
Attacker Front-run with Donation of 20000 USDT	20000e6 + 1	1 (owned by attacker)
Victim's Actual Deposit	20000e6+1 (From attacker) + 20000e6 (From Victim) = 40000e6 + 1	1*20000e6/20000e6 +1 = 0 (Victim does not get shares) **Total shares = 1** (initially owned by Attacker)

INFLATION ATTACK

This attack is also sometimes referred to as the “First-minter problem”

- It is called the First-minter Problem because only the first minter has the ability to perform this attack.
- Inflation attack causes the trustlessness of vault protocols to be threatened and provide less confidence for users to engage in depositing actions with newly created vaults.

COMMON SOLUTION

- One common solution to the problem is to create “Dead Shares”
  - A solution used in UniswapV2 Protocol
- During the initial mint (or deposit), a fixed amount of dead shares are minted to address(0)
  - These shares are irretrievable permanently
  - Increases the complexity of the attack
  - Removes economic incentives
- Although there are no economic incentives, griefing opportunity is still present.

```
1  function deposit(uint256 assets) public {
2      uint256 shares = convertToShares(assets);
3
4      if (totalShares() == 0) {
5          _mint(address(0), NUMBER_OF_DEAD_SHARES);
6          shares -= NUMBER_OF_DEAD_SHARES;
7      }
8
9      asset.transferFrom(msg.sender, address(this), assets);
10     _mint(msg.sender, shares);
11 }
```

INFLATION ATTACK

Attack scenario:

1. The attacker back-runs a transaction of an ERC4626 pool creation.
2. The attacker mints 1,000 shares: deposit(1000). Thus, totalAsset()==1000, totalSupply()==1000. Note that the balanceOf(attacker) == 0 and balanceOf(address(0)) == 1000 in this example.
3. The attacker front-runs the deposit of the victim who wants to deposit 20,000 USDT (20,000.000000).
4. The attacker inflates the denominator right in front of the victim: asset.transfer(20_000_000e6). Now totalAsset() == 20_000_000e6 + 1000, totalSupply() == 1000.
5. Next, the victim's tx takes place. The victim gets 1000 * 20_000e6 / (20_000_000e6 + 1000) == 0 shares. The victim gets zero shares, losing their deposit to the pool.
6. Thus, the attacker burns any deposit of the victim, but spends a thousand times more money to do so.

	Assets	Shares
Initial State	0	0
After Attacker Deposit	1000	Dead shares = 1000, Attacker Shares = 0
Victim Wants to Deposit 20000 USDT	Expected Assets = 20000e6 + 1000	Expected Shares = 20000e6*1000 /1000 = 20000e6
Attacker Front-run with Donation of 20000000 USDT	20000000e6 + 1	1000 (Dead Shares)
Victim's Actual Deposit	20000e6 + 20000000e6+1000 = 20020000e6 + 1000	1000*20000e6/20000000e6 +1000 = 0 (Victim does not get shares) Total shares = 1000 (Dead shares)

ERC3156 FLASH LOAN INTERFACE

ERC3516 is an interface standard for flash loans in Ethereum-based decentralized finance (DeFi) applications.

- Flash loan is an uncollateralized loan
  - It must be repaid within the same blockchain transaction
- Any one with smart contract knowledge can access it
- Loan amount could reach as high as millions USD with minimal interest rate
  - Aave Lending Protocol has a 0.09% and DyDx provides for free.

CODE OVERVIEW

01 - FLASHLOAN

```solidity
interface IERC3156FlashLender {

    /**
     * @dev The amount of currency available to be lent.
     * @param token The loan currency.
     * @return The amount of `token` that can be borrowed.
     */
    function maxFlashLoan(
        address token
    ) external view returns (uint256);

    /**
     * @dev The fee to be charged for a given loan.
     * @param token The loan currency.
     * @param amount The amount of tokens lent.
     * @return The amount of `token` to be charged for the loan, on top of the returned principal.
     */
    function flashFee(
        address token,
        uint256 amount
    ) external view returns (uint256);

    /**
     * @dev Initiate a flash loan.
     * @param receiver The receiver of the tokens in the loan, and the receiver of the callback.
     * @param token The loan currency.
     * @param amount The amount of tokens lent.
     * @param data Arbitrary data structure, intended to contain user-defined parameters.
     */
    function flashLoan(
        IERC3156FlashBorrower receiver,
        address token,
        uint256 amount,
        bytes calldata data
    ) external returns (bool);
}
```

VULNERABILITIES

Although we have mentioned that the service provider of a flashloan does not have any risk, there are actually a lot of vulnerabilities if the flashloan is not designed well within a contract.

- Flashloan is a service that contains the following properties:
  - Balance checking
    - inaccuracy of balance checking may lead to loss of balance
  - Callback functions are used
    - Leads to reentrancy attacks
    - May cause other vulnerabilities

LOW-LEVEL CALL VULNERABILITY

01 - FUNCTIONCALL

- Flash Loan Function's usual procedure
  a. Optimistically transfer you your requested amount of tokens
  b. Call-back function for you to use the tokens
  c. Check whether tokens are repaid

```
function flashLoan(
        uint256 amount,
        address borrower,
        address target, //usdtAddress
        bytes calldata data //abi.encode("approve()...")
) external nonReentrant returns (bool) {
        uint256 balanceBefore = token.balanceOf(address(this));
        token.transfer(borrower, amount);
        target.functionCall(data); //From Address.sol

        if (token.balanceOf(address(this)) < balanceBefore)
                revert RepayFailed();

        return true;
}
```

LOW-LEVEL CALL VULNERABILITY

01 - FUNCTIONCALL

- By bringing in the token address as the variable “target” and the abi.encode(“approve()...”) as the data, we can lead the flashloan contract to approve us for any amount
  - By doing so, we can just call transferFrom afterwards and steal all the funds within the smart contract.

```
function functionCallWithValue(address target, bytes memory data, uint256 value) internal
    if (address(this).balance < value) {
        revert Errors.InsufficientBalance(address(this).balance, value);
    }
    (bool success, bytes memory returndata) = target.call{value: value}(data);
    return verifyCallResultFromTarget(target, success, returndata);
}
```

```
function flashLoan(
        uint256 amount,
        address borrower,
        address target, //usdtAddress
        bytes calldata data //abi.encode("approve()...")
) external nonReentrant returns (bool) {
        uint256 balanceBefore = token.balanceOf(address(this));
        token.transfer(borrower, amount);
        target.call(data); //From Address.sol
        //usdtAddress.approve();

        if (token.balanceOf(address(this)) < balanceBefore)
                revert RepayFailed();

        return true;
}
```

https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/utils/Address.sol#L75

SIMPLE SOLUTION

- We can take a look at Aave's implementation of a flash loan
- Instead using low-level for callback, we design a flash loan receiver interface instead
  - Only smart contracts can call flash loan
  - Must inherit this interface and implement the function executeOperation()
  - Cannot use abi.encode to call any malicious functions

```solidity
interface IFlashLoanSimpleReceiver {
  /**
   * @notice Executes an operation after receiving the flash-borrowed asset
   * @dev Ensure that the contract can return the debt + premium, e.g., has
   *      enough funds to repay and has approved the Pool to pull the total amount
   * @param asset The address of the flash-borrowed asset
   * @param amount The amount of the flash-borrowed asset
   * @param premium The fee of the flash-borrowed asset
   * @param initiator The address of the flashloan initiator
   * @param params The byte-encoded params passed when initiating the flashloan
   * @return True if the execution of the operation succeeds, false otherwise
   */
  function executeOperation(
    address asset,
    uint256 amount,
    uint256 premium,
    address initiator,
    bytes calldata params
  ) external returns (bool);

  function ADDRESSES_PROVIDER() external view returns (IPoolAddressesProvider);

  function POOL() external view returns (IPool);
}
```

https://docs.aave.com/developers/guides/flash-loans

RE-ENTRANCY VULNERABILITY

- Most Flashloan services exist in pool or bank contracts where there are other functionalities

- Attack scenario
  - First call flashLoan
  - During executeOperation() call-back, call deposit
  - User bank balance increases and bank contract balance = balanceBefore

- Solution: **Prevent reentrancy** or **Call transferFrom** instead of balance checking

```
1  contract Bank {
2      address tokenAddress; //assume tokenAddress set during constructor
3      mapping (address => uint256) balances;
4
5      //  We can use transferFrom to handle deposit actions
6      function deposit(uint256 amount) public {
7          IERC20(tokenAddress).transferFrom(msg.sender, address(this), amount);
8          balance[msg.sender] += amount;
9      }
10
11     //  We can use transfer to handle withdrawal actions
12     function withdraw(uint256 amount) public {
13         require(balances[msg.sender] >= amount, "Insufficient Balance");
14         IERC20(tokenAddress).transfer(msg.sender, amount);
15         balance[msg.sender] -= amount;
16     }
17
18     function flashLoan(
19         uint256 amount,
20         address borrower,
21         address target,
22         bytes calldata data
23     ) external nonReentrant returns (bool) {
24         uint256 balanceBefore = IERC20(tokenAddress).balanceOf(address(this));
25         IERC20(tokenAddress).transfer(borrower, amount);
26         target.executeOperation(data);
27
28         if (IERC20(tokenAddress).balanceOf(address(this)) < balanceBefore)
29                 revert RepayFailed();
30
31         return true;
32     }
33 }
```

REENTRANCY GUARD/TRANSFERFROM?

- Both solutions are adequate however poses their own unique security concerns

1. Reentrancy guard limits function and contract flexibility
2. transferFrom instead of balance checking may cause balance loss on minor tokens with “fee-on-transfer” implemented