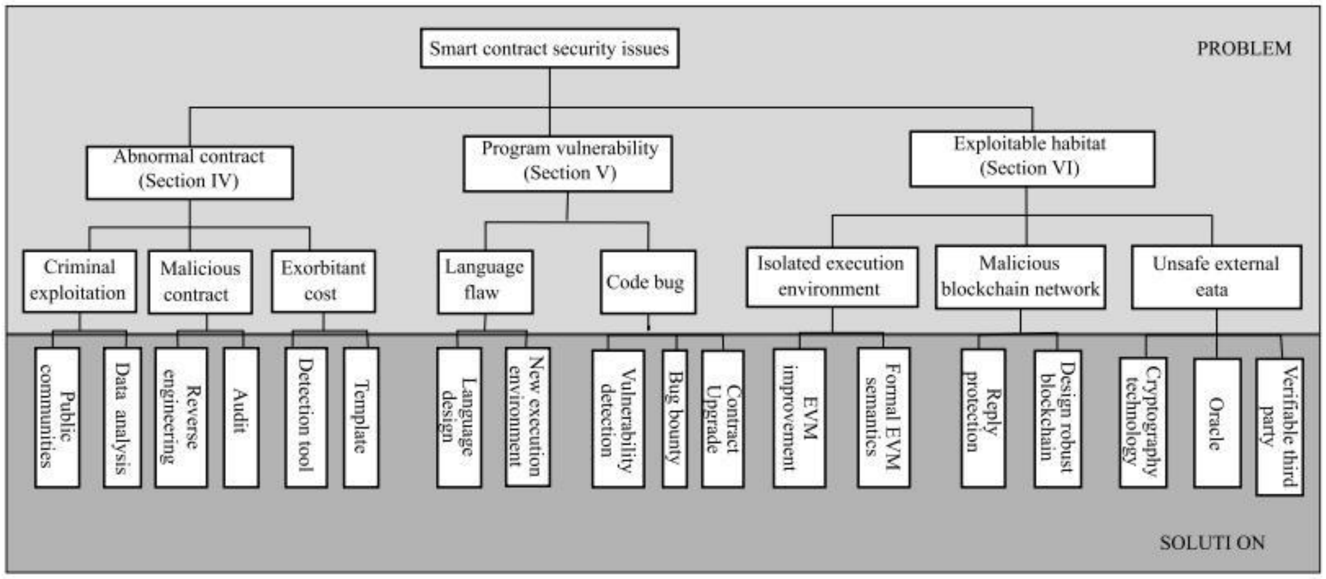
The documentation on common attack prevention – Martino Agostini (martino.agostini@gmail.com)

Blockchain platforms and languages for writing smart contracts are becoming increasingly popular. However, smart contracts and blockchain applications are developed through non-standard software life cycles. For instance, delivered applications can hardly be updated or bugs resolved by releasing a new version of the software (Vacca et al. 2021).

More focus needs to be dedicated to Smart contract security in this scenario. This emerging research area deals with security issues arising from the execution of smart contracts in a blockchain system. Generally, a smart contract is a piece of executable code that automatically runs on the blockchain to enforce an agreement preset between parties involved in the transaction. To address these vulnerabilities, we examine recent advances in smart contract security spanning four development phases: 1) security design, 2) security implementation, 3) testing before deployment, and 4) monitoring and analysis. Finally, we outline emerging challenges and opportunities in smart contract security for blockchain engineers and researchers (Huang et al. 2019).

An interesting snapshot of the complexity of the subject is reported in the paper “Ethereum smart contract security research: survey and future research opportunities” (Wang et al. 2021).



Source: Zeli WANG et al.

For this project, we can summarize the secure development in different areas that cover: the danger of token integration , strategy for security access control, and future implementation of access to prices oracle and more.

Following the best practice reported in the https://consensys.github.io/smart-contract-best-practices/known_attacks/

Prevent transferring tokens to the 0x0 address¶

At the time of writing, the "zero" address (0x0000000000000000000000000000000000000000) holds tokens with a value of more than 80$ million.

Prevent transferring tokens to the contract address¶

Consider also preventing the transfer of tokens to the same address of the smart contract.

An example of the potential for loss by leaving this open is the EOS token smart contract where more than 90,000 tokens are stuck at the contract address.

Example¶

An example of implementing both the above recommendations would be to create the following modifier; validating that the "to" address is neither 0x0 nor the smart contract's own address:

```
modifier validDestination( address to ) {
    require(to != address(0x0));
    require(to != address(this) );
    _;
}
```

The modifier should then be applied to the "transfer" and "transferFrom" methods:

```
function transfer(address _to, uint _value)
    validDestination(_to)
    returns (bool)
{
    (... your logic ...)
}

function transferFrom(address _from, address _to, uint _value)
    validDestination(_to)
    returns (bool)
{
    (... your logic ...)
}
```

What next ?

In the future implementation, I will One of the significant dangers of calling external contracts is taking over the control flow.

To prevent Reentrancy on a Single Function

The best way to prevent this attack is to make sure you don't call an external function until you've done all the internal work you need to do:

```
mapping (address => uint) private userBalances;

function withdrawBalance() public {
    uint amountToWithdraw = userBalances[msg.sender];
    userBalances[msg.sender] = 0;
    (bool success, ) = msg.sender.call.value(amountToWithdraw)(""); // The user's
balance is already 0, so future invocations won't withdraw anything
    require(success);
}
```

To prevent Cross-function Reentrancy¶

An attacker may also do a similar attack using two different functions that share the same state.

Pitfalls in Reentrancy Solutions

Consensys recommended finishing all internal work (ie. state changes) first, and only then calling the external function. In particular, it is suggested not only to avoid calling external functions too soon but also avoid calling functions that call external functions

Last but not least, there are a few high-risk areas that need to be considered in the following categories of **front-running** attacks: Displacement, Insertion, and Suppression

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

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

             /**
              * @dev Atomically decreases the allowance granted to `spender` by the caller.
              *
```

```
     * This is an alternative to {approve} that can be used as a mitigation for
     * problems described in {IERC20-approve}.
     *
     * Emits an {Approval} event indicating the updated allowance.
     *
     * Requirements:
     *
     * - `spender` cannot be the zero address.
     * - `spender` must have allowance for the caller of at least
     * `subtractedValue`.
     */
    function decreaseAllowance(address spender, uint256 subtractedValue) public virtual
returns (bool) {
        uint256 currentAllowance = _allowances[_msgSender()][spender];
        require(currentAllowance >= subtractedValue, "ERC20: decreased allowance below
zero");
        unchecked {
            _approve(_msgSender(), spender, currentAllowance - subtractedValue);
        }

        return true;
    }
```

To sum up Consensys suggest this **Token Checklist** regarding ERC20

Token	Feature	Known Vulnerabilities	Resources	Examples
ERC20	**Allowance**	Double withdrawal (front-running)	Resolving the Multiple Withdrawal Attack on ERC20 Tokens	
	decimals()	The decimals can be more than 18		YamV2 has 24 decimals
		Not accounting for the tokens that try to prevent multiple withdrawal attack	Perpetual Protocol Audit issue 3.12	

		Unprotected *transferFrom()*	Bancor Network Hack 2020 - 1inch	
	External Calls	Unchecked Call Return Value	Unchecked call return value	
		DoS with unexpected revert	DoS with unexpected revert	
	Transfers	Might return False instead of Revert		
		Missing return value	Missing return value bug — At least 130 tokens affected	
	BalanceOf()	Internal Accounting discrepancy with the Actual Balance	aToken Withdrawal Vulnerability	aToken
	Blacklistable	Blacklisted addresses cannot receive or send tokens	CENTRE appears to have blacklisted an address holding USDC for the first time	USDC (FiatToken)
	Mintable / Burnable	TotalSupply can change by trusted actors		
	Pausable	All functionalities can be paused by trusted actors		

Source : https://consensys.net/diligence/blog/2020/11/token-interaction-checklist/

An **additional Token integration checklist** is available on

- **https://github.com/crytic/building-secure-contracts/blob/master/development-guidelines/token_integration.md**
- **Resolving the Multiple Withdrawal Attack on ERC20 Tokens**

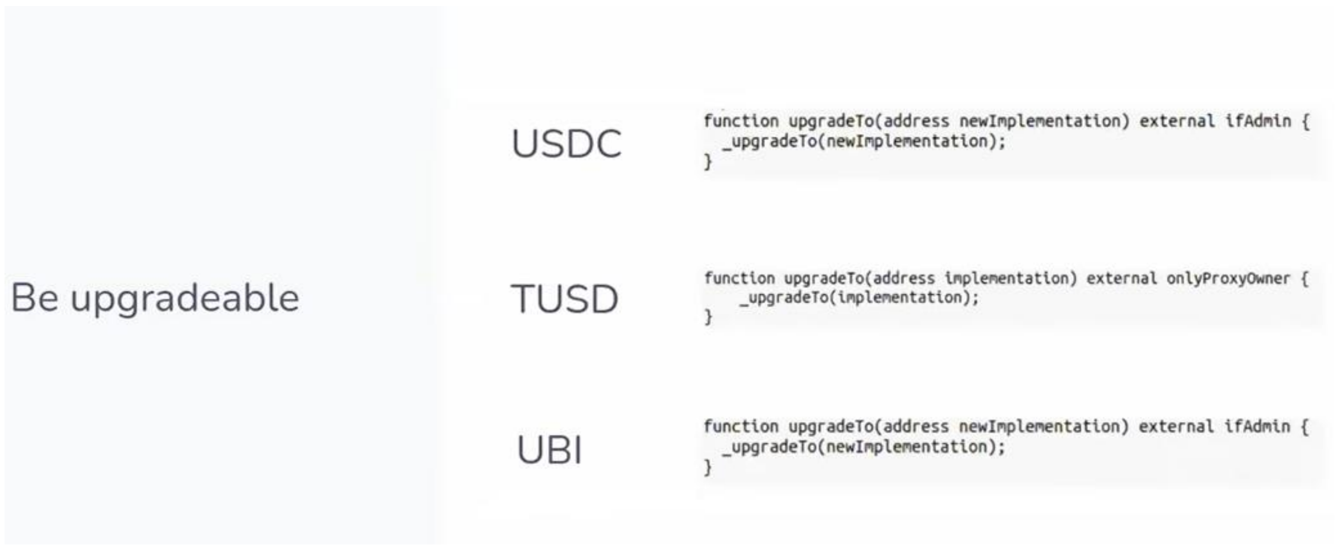
Few final considerations

The first is that introducing a token in your system could even raise an economic risk that needs to be adequately considered.

The second is to monitor and accordingly plan the upgradeable token in your applications but even consider the cryptocurrency (example reported below) that could be utilized with it. This allows changing **the contract code while preserving the state, balance, and address**.

Consequently, the Governance aspect and the awareness to the user on this peculiarity are key for the user and the auditor to assess their evaluation appropriately.

Last but not least, while any smart contract can be made upgradeable, some restrictions of the Solidity language need to be worked around. These come up when writing both the initial version of the contract and the version we'll upgrade it to.



Be upgradeable

USDC

```
function upgradeTo(address newImplementation) external ifAdmin {
  _upgradeTo(newImplementation);
}
```

TUSD

```
function upgradeTo(address implementation) external onlyProxyOwner {
    _upgradeTo(implementation);
}
```

UBI

```
function upgradeTo(address newImplementation) external ifAdmin {
  _upgradeTo(newImplementation);
}
```

More info on https://docs.openzeppelin.com/learn/upgrading-smart-contracts