Workshop #1
Secure Development Series

07/15 - 12PM PST / 7PM UTC

The Dangers of Token Integration

Martin Abbatemarco
Security Researcher at OpenZeppelin

Series of sessions

Secure Development

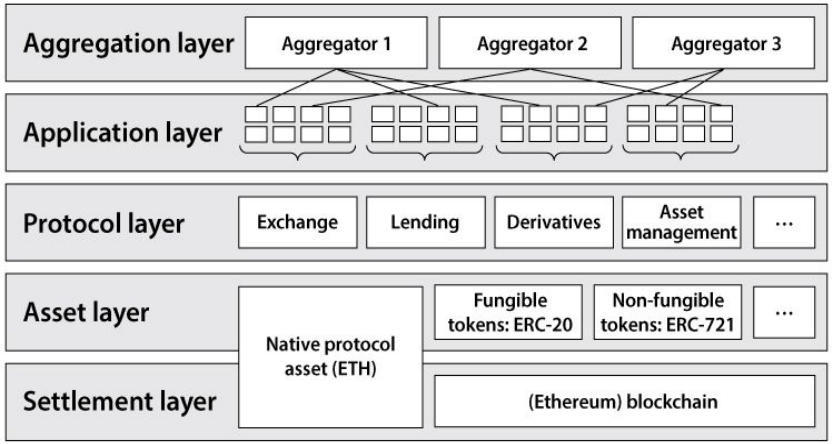
The dangers of token integration

Strategies for secure access controls

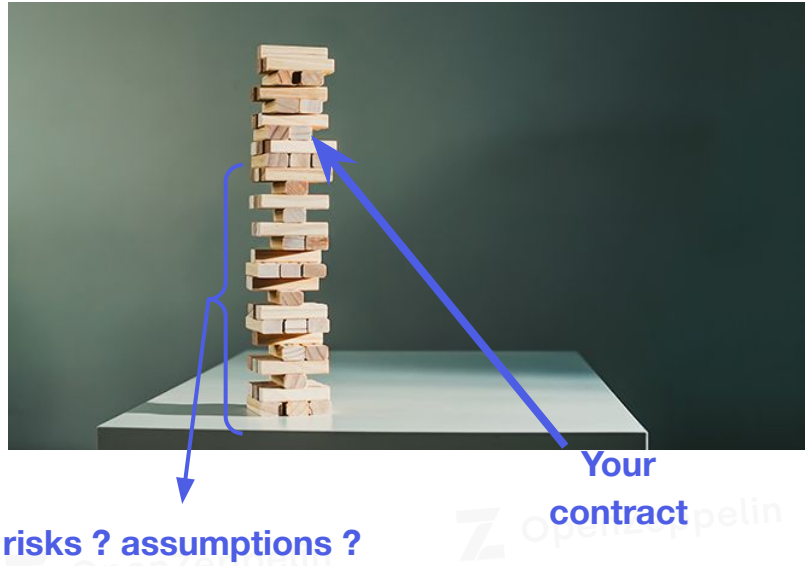
The dangers of price oracles

and more!

Composability



Decentralized Finance: On Blockchain- and Smart Contract-Based Financial Markets by Fabian Schär



some initial triggers

```solidity
interface IERC20 {
    function transferFrom(address sender, address recipient, uint256 amount) external returns (bool);
}

contract ArbitraryTokenHandler {
    IERC20 immutable token;
    uint256 totalDeposits;
    mapping (address => uint256) deposits;

    constructor(address _token) {
        token = IERC20(_token);
    }

    function deposit(uint256 amount) external {
        token.transferFrom(msg.sender, address(this), amount);
        deposits[msg.sender] += amount;
        totalDeposits += amount;
    }
}
```

Does it even compile ?

What Solidity version ? Isn't that overflowing ?

No docstrings - what's this *supposed* to do ?

No functions to withdraw tokens ?

No visibility on state variables ?

No events ?

Assuming previous approval ?

Not checking return value ?

Reentrancy ?

Works with *any* token ?

...

Addresses can be blocked

Compromised blockers may put your system **at risk**

USDC

```
contract FiatTokenV1 is AbstractFiatTokenV1, Ownable, Pausable, Blacklistable {

function transfer(address to, uint256 value)
    external
    override
    whenNotPaused
    notBlacklisted(msg.sender)
    notBlacklisted(to)
    returns (bool)
{
    _transfer(msg.sender, to, value);
    return true;
}
```

9. blacklister

0x5db0115f3b72d19cea34dd697cf412ff86dc7e1b

```
if IUSDC(usdcAddress).isBlacklisted(someAddress) {
    // logic for address blocked
} else {
    // logic address allowed
}
```

USDT

```
contract TetherToken is Pausable, StandardToken, BlackList {

function destroyBlackFunds (address _blackListedUser) public onlyOwner {

function transfer(address _to, uint _value) public whenNotPaused {
    require(!isBlackListed[msg.sender]);
```

14. owner

0xc6cde7c39eb2f0f0095f41570af89efc2c1ea828

```
if IUSDT(usdtAddress).isBlackListed(someAddress) {
    // logic for address blocked
} else {
    // logic address allowed
}
```

Can be paused

USDC

```
function pause() external onlyPauser {
    paused = true;
    emit Pause();
}
function transfer(address to, uint256 value)
    external
    override
    whenNotPaused
```

USDT

```
function pause() onlyOwner whenNotPaused public {
  paused = true;
  Pause();
}
function transfer(address _to, uint _value) public whenNotPaused {
```

BNT

```
function disableTransfers(bool _disable) public ownerOnly {
    transfersEnabled = !_disable;
}
modifier transfersAllowed {
    assert(transfersEnabled);
    _;
}
function transfer(address _to, uint256 _value) public transfersAllowed returns (bool success) {
```

Send / receive hooks

ERC777 → hooks on sender & receiver

ERC721 → hook on receiver

ERC1363 → hook on receiver / spender

and more!

Send / receive hooks

For example, on transfers:

1. call sender
2. modify balances
3. call receiver

Send / receive hooks

```
15      /* VULNERABLE CODE */
16
17      function supply(uint256 amount) public {
18          uint256 newBalance = amount + deposits[msg.sender];
19
20          /*
21           * behind the scenes:
22           * 1. executes hook on attacker --> transfers execution to attacker!
23           * 2. updates balances
24           * 3. executes hook on receiver
25           */
26          token.transferFrom(msg.sender, address(this), amount);
27
28          deposits[msg.sender] = newBalance;
29      }
30
31      function withdraw() public {
32          uint256 amountToWithdraw = deposits[msg.sender];
33
34          deposits[msg.sender] = 0;
35
36          token.transfer(msg.sender, amountToWithdraw);
37      }
38  }
```

Read more:

- imBTC hacks
- https://github.com/OpenZeppelin/exploit-uniswap

Prevent transfers

- to token contract
- to zero address
- to caller's address
- of value zero

LINK

```
modifier validRecipient(address _recipient) {
  require(_recipient != address(0) && _recipient != address(this));
  _;
}

function transfer(address _to, uint _value)
  public
  validRecipient(_to)
```

BNB

```
function transfer(address _to, uint256 _value) {
    if (_to == 0x0) throw;
    if (_value <= 0) throw;
```

UNI

```
require(dst != address(0), "Uni::_transferTokens: cannot transfer to the zero address");
```

OpenZeppelin's ERC20

```
function _transfer(
    address sender,
    address recipient,
    uint256 amount
) internal virtual {

    require(recipient != address(0), "ERC20: transfer to the zero address");
```

Compound's cTokens

```
/* Do not allow self-transfers */
if (src == dst) {
    return fail(Error.BAD_INPUT, FailureInfo.TRANSFER_NOT_ALLOWED);
}
```

Take fees
on transfers

USDT

```
function transferFrom(address _from, address _to, uint _value) public
    uint sendAmount = _value.sub(fee);
    balances[_from] = balances[_from].sub(_value);
    balances[_to] = balances[_to].add(sendAmount);
```

STA

```
function transferFrom(address from, address to, uint256 value) public returns (bool) {
    _balances[from] = _balances[from].sub(value);

    uint256 tokensToBurn = cut(value);
    uint256 tokensToTransfer = value.sub(tokensToBurn);

    _balances[to] = _balances[to].add(tokensToTransfer);
    _totalSupply = _totalSupply.sub(tokensToBurn);
```

was used to steal from Balancer pools

How does **Uniswap** v2 tackle this ?

Take fees
on transfers

```
function addLiquidity(
    address tokenA,
    address tokenB,
    uint amountADesired,
    uint amountBDesired,
    uint amountAMin,
    uint amountBMin,
    address to,
    uint deadline
) external virtual override ensure(deadline) returns (uint amountA, uint amountB, uint liquidity) {
    (amountA, amountB) = _addLiquidity(tokenA, tokenB, amountADesired, amountBDesired, amountAMin, amountBMin);
    address pair = UniswapV2Library.pairFor(factory, tokenA, tokenB);
    TransferHelper.safeTransferFrom(tokenA, msg.sender, pair, amountA);
    TransferHelper.safeTransferFrom(tokenB, msg.sender, pair, amountB);
    liquidity = IUniswapV2Pair(pair).mint(to);
}
```

```
function mint(address to) external lock returns (uint liquidity) {
    (uint112 _reserve0, uint112 _reserve1,) = getReserves(); // gas savings
    uint balance0 = IERC20(token0).balanceOf(address(this));
    uint balance1 = IERC20(token1).balanceOf(address(this));
```

Just accounting for what it actually received

How does **Compound** tackle this ?

Take fees
on transfers

```
function doTransferIn(address from, uint amount) internal returns (uint) {
    EIP20NonStandardInterface token = EIP20NonStandardInterface(underlying);
    uint balanceBefore = EIP20Interface(underlying).balanceOf(address(this));
    token.transferFrom(from, address(this), amount);

    bool success;
    assembly {
        switch returndatasize()
            case 0 {                      // This is a non-standard ERC-20
                success := not(0)         // set success to true
            }
            case 32 {                     // This is a compliant ERC-20
                returndatacopy(0, 0, 32)
                success := mload(0)       // Set `success = returndata` of external call
            }
            default {                     // This is an excessively non-compliant ERC-20, revert.
                revert(0, 0)
            }
    }
    require(success, "TOKEN_TRANSFER_IN_FAILED");

    // Calculate the amount that was *actually* transferred
    uint balanceAfter = EIP20Interface(underlying).balanceOf(address(this));
    require(balanceAfter >= balanceBefore, "TOKEN_TRANSFER_IN_OVERFLOW");
    return balanceAfter - balanceBefore;   // underflow already checked above, just subtract
}
```

1. Check balance *before* transfer
2. Execute transfer
3. Check balance *after* transfer
4. Only account for difference
(what the contract actually received)

Different ways of signaling failure / success

```
function transfer(address _to, uint _value) returns (bool) {
    //Default assumes totalSupply can't be over max (2^256 - 1).
    if (balances[msg.sender] >= _value && balances[_to] + _value >= balances[_to]) {
        balances[msg.sender] -= _value;
        balances[_to] += _value;
        Transfer(msg.sender, _to, _value);
        return true;
    } else { return false; }
}
```

ZRX

```
function transferFrom(address _from, address _to, uint256 _value) returns (bool success) {
function transfer(address _to, uint256 _value) {
```

BNB

```
function transfer(address _to, uint _value) public onlyPayloadSize(2 * 32) {
```

USDT

```
function transfer(address _to, uint _value) public returns (bool) {
  uint fee = calcFee(_value);
  uint sendAmount = _value.sub(fee);

  super.transfer(_to, sendAmount);
  if (fee > 0) {
    super.transfer(owner(), fee);
  }
}
```

Gold Tether

How to handle *most* of these cases ?

Using **wrappers** for a more consistent behavior

Different ways of signaling failure / success

SafeERC20

```
import "@openzeppelin/contracts/token/ERC20/utils/SafeERC20.sol";
```

Wrappers around ERC20 operations that throw on failure (when the token contract returns false). Tokens that return no value (and instead revert or throw on failure) are also supported, non-reverting calls are assumed to be successful. To use this library you can add a `using SafeERC20 for IERC20;` statement to your contract, which allows you to call the safe operations as `token.safeTransfer(…)`, etc.

FUNCTIONS

```
safeTransfer(token, to, value)
safeTransferFrom(token, from, to, value)
safeApprove(token, spender, value)
safeIncreaseAllowance(token, spender, value)
safeDecreaseAllowance(token, spender, value)
```

docs.openzeppelin.com/contracts/4.x/api/token/erc20#SafeERC20

Be flash-loanable

Don’t assume an account cannot have a significantly large balance of tokens.

Test for extreme values in deposits, swaps, etc.

In some cases token snapshots from a previous block help.

```
require(
    comp.getPriorVotes(msg.sender, sub256(block.number, 1)) > proposalThreshold,
    "GovernorBravo::propose: proposer votes below proposal threshold"
);
```

Don’t assume an account cannot have a absurdly large balance of tokens.

Be flash-mintable

ERC20FlashMint

```
import "@openzeppelin/contracts/token/ERC20/extensions/draft-ERC20FlashMint.sol";
```

Implementation of the ERC3156 Flash loans extension, as defined in ERC-3156.

Adds the `flashLoan` method, which provides flash loan support at the token level. By default there is no fee, but this can be changed by overriding `flashFee`.

Available since v4.1.

FUNCTIONS

```
maxFlashLoan(token)
flashFee(token, amount)
flashLoan(receiver, token, amount, data)
```

Have huge total supply

(breaking interoperability)

Uniswap V2 limits pool balances to `2**112-1`

```
// update reserves and, on the first call per block, price accumulators
function _update(uint balance0, uint balance1, uint112 _reserve0, uint112 _reserve1) private {
    require(balance0 <= uint112(-1) && balance1 <= uint112(-1), 'UniswapV2: OVERFLOW');
```

this limit is due to use of fixed point math libraries

Modify accounts balance

outside transfer operations

aTokens

```
/**
 * @dev Calculates the balance of the user: principal balance + interest generated by the principal
 * @param user The user whose balance is calculated
 * @return The balance of the user
 **/
function balanceOf(address user)
  public
  view
  override(IncentivizedERC20, IERC20)
  returns (uint256)
{
  return super.balanceOf(user).rayMul(_pool.getReserveNormalizedIncome(_underlyingAsset));
}
```

UBI

```
function balanceOf(address _human) public view returns (uint256) {
  return getAccruedValue(_human).add(balance[_human]);
}
```

AMPL

```
function rebase(uint256 epoch, int256 supplyDelta)
    external
    onlyMonetaryPolicy
    returns (uint256)
{
    if (supplyDelta == 0) {
        emit LogRebase(epoch, _totalSupply);
        return _totalSupply;
    }

    if (supplyDelta < 0) {
        _totalSupply = _totalSupply.sub(uint256(supplyDelta.abs()));
    } else {
        _totalSupply = _totalSupply.add(uint256(supplyDelta));
    }

    if (_totalSupply > MAX_SUPPLY) {
        _totalSupply = MAX_SUPPLY;
    }

    _gonsPerFragment = TOTAL_GONS.div(_totalSupply);
```

```
function balanceOf(address who) external view override returns (uint256) {
    return _gonBalances[who].div(_gonsPerFragment);
}
```

Not expose `decimals`

Not expose `name`

Not expose `symbol`

(and still be ERC20-compliant)

name

Returns the name of the token - e.g. `"MyToken"`.

OPTIONAL - This method can be used to improve usability, but interfaces and other contracts MUST NOT expect these values to be present.

```
function name() public view returns (string)
```

symbol

Returns the symbol of the token. E.g. “HIX”.

OPTIONAL - This method can be used to improve usability, but interfaces and other contracts MUST NOT expect these values to be present.

```
function symbol() public view returns (string)
```

decimals

Returns the number of decimals the token uses - e.g. `8`, means to divide the token amount by `100000000` to get its user representation.

OPTIONAL - This method can be used to improve usability, but interfaces and other contracts MUST NOT expect these values to be present.

```
function decimals() public view returns (uint8)
```

eips.ethereum.org/EIPS/eip-20#specification

```
// IERC20Standard.decimals() will revert if the collateral contract has not implemented the decimals() method,
// which is possible since the method is only an OPTIONAL method in the ERC20 standard:
// https://eips.ethereum.org/EIPS/eip-20#methods.
function _getSyntheticDecimals(address _collateralAddress) public view returns (uint8 decimals) {
    try IERC20Standard(_collateralAddress).decimals() returns (uint8 _decimals) {
        return _decimals;
    } catch {
        return 18;
    }
}
```

https://github.com/UMAprotocol/protocol/blob/master/packages/core/contracts/financial-templates/expiring-multiparty/PricelessPositionManager.sol

Don’t assume all tokens have 18 decimals

Not have
18 decimals

DAI → 18 decimals

USDC → 6 decimals

YAM-v2 → 24 decimals

cTokens → 8 decimals

WBTC → 8 decimals

Some cases of handling decimals:

- `github.com/compound-finance/open-oracle/blob/master/contracts/Uniswap/UniswapAnchoredView.sol`
- `github.com/aave/protocol-v2/blob/master/contracts/adapters/BaseUniswapAdapter.sol`

OpenZeppelin's ERC20

```
148     function transferFrom(
149         address sender,
150         address recipient,
151         uint256 amount
152     ) public virtual override returns (bool) {
153         _transfer(sender, recipient, amount);
154 
155         uint256 currentAllowance = _allowances[sender][_msgSender()];
156         require(currentAllowance >= amount, "ERC20: transfer amount exceeds allowance");
157         unchecked {
158             _approve(sender, _msgSender(), currentAllowance - amount);
159         }
160 
161         return true;
162     }
```

Have "infinite" approval

(does not decrease on `transferFrom`)

vs.

DAI

```
function transferFrom(address src, address dst, uint wad)
    public returns (bool)
{
    require(balanceOf[src] >= wad, "Dai/insufficient-balance");
    if (src != msg.sender && allowance[src][msg.sender] != uint(-1)) {
        require(allowance[src][msg.sender] >= wad, "Dai/insufficient-allowance");
        allowance[src][msg.sender] = sub(allowance[src][msg.sender], wad);
    }
```

Approval of 2**256-1 is considered permanent and never decreases

May not guard against the ERC20 allowance front-running attack

OpenZeppelin Contract's ERC20 implements functions to prevent this attack

`increaseAllowance``(address spender, uint256 addedValue) → bool` public #

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.

`decreaseAllowance``(address spender, uint256 subtractedValue) → bool` public #

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`.

https://docs.openzeppelin.com/contracts/4.x/api/token/erc20

The ERC20 allowance front-running attack by Vladimirov & Khovratovich
https://docs.google.com/document/d/1YLPtQxZu1UAvO9cZ1O2RPXBbT0mooh4DYKjA_jp-RLM/edit

Disallow approvals

when already positive

```
function approve(address _spender, uint _value) public onlyPayloadSize(2 * 32) {

    // To change the approve amount you first have to reduce the addresses`
    //  allowance to zero by calling `approve(_spender, 0)` if it is not
    //  already 0 to mitigate the race condition described here:
    //  https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729
    require(!((_value != 0) && (allowed[msg.sender][_spender] != 0)));

    allowed[msg.sender][_spender] = _value;
    Approval(msg.sender, _spender, _value);
}
```

USDT

Does not allow approving tokens when approval is non-zero.

This is a way to prevent the ERC20 allowance front-running attack, but introducing unexpected behavior might break interoperability.

Consider special cases in off-chain monitoring

Not emit expected events

```
function mint(address guy, uint wad) public auth stoppable {
    _balances[guy] = add(_balances[guy], wad);
    _supply = add(_supply, wad);
    Mint(guy, wad);
}
function burn(address guy, uint wad) public auth stoppable {
    if (guy != msg.sender && _approvals[guy][msg.sender] != uint(-1)) {
        _approvals[guy][msg.sender] = sub(_approvals[guy][msg.sender], wad);
    }

    _balances[guy] = sub(_balances[guy], wad);
    _supply = sub(_supply, wad);
    Burn(guy, wad);
}
```

MKR

Be upgradeable

USDC

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

TUSD

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

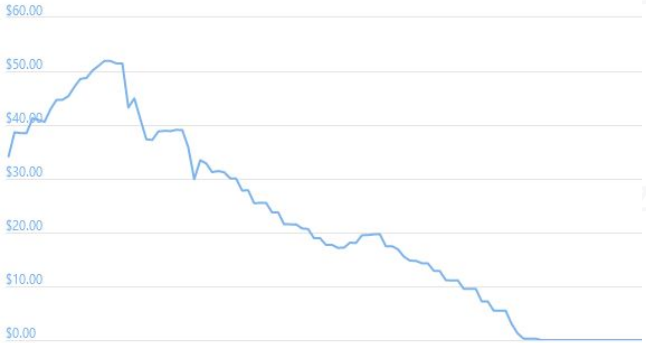
UBI

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

Might need allowing interactions only with known implementations

```
require(implementations[gem.implementation()] == 1, "GemJoin6/implementation-invalid");
```

And we haven’t even discussed economic risks



On token integration

Closing thoughts

1. Zero trust mindset. Tokens are arbitrary code.
2. Healthy distrust for standards.
3. Consider list of allowed tokens. Verify behavior. Analyze risks.
4. See how experienced players do it. Learn from their code.
5. Document your trust assumptions.

On token integration

Where do I learn more ?

(because yes, there's probably more)

- ➔ https://github.com/d-xo/weird-erc20
- ➔ Token integration checklist by Trail of Bits
- ➔ Token interaction checklist by Consensys Diligence
- ➔ https://github.com/sec-bit/awesome-buggy-erc20-tokens

Series of sessions

Secure Development

The dangers of token integration 

Strategies for secure access controls

The dangers of price oracles

and more!

In the meantime…

docs.openzeppelin.com/defender/advisor

Thanks!

Learn more

openzeppelin.com
defender.openzeppelin.com
blog.openzeppelin.com
forum.openzeppelin.com

Contact

@tinchoabbate
tincho@openzeppelin.com