Staking Contract Design

## General design

The first version of the staking contract should be ONE POOL similar to the sushiswap pool:

* User can stake and withdraw
* Reward is calculated for each block using the sushiswap design (e.g. update each user’s pending rewards each time before staking changes)
* Reward rate should be configurable
* The contract should be able to accept EVM as Reward source
* If possible, the staking pool and reward pool should be separated
* Reward is paid each time user deposit or withdraw
* Claim reward is done via deposit 0 (or withdraw 0 depends on implementation)
* Emergency withdraw that will skip the reward logic

Extra features needed:

* An allow list controlled by admin that only user in the list can stake.
* One should still be able to perform emergency withdraw if kicked out of the allow list
* User can set an url associated with his account, proxy will read this url as upstream address
* Staking is locked after deposit for a configurable amount of time. Only emergency withdraw is allowed during lock period.
* **Being able to upgrade**

## Some clarification regarding emergency withdraw

Emergency withdraw is designed as an emergency exit for user who want their fund back without caring about the rewards. It is meant to be a minimal robust API so that even when there’s something seriously wrong with the contract, the user can still get their money back.

Example: (code taken from pancake)

We do not necessary go for the same design, just use it as an example.

// Withdraw without caring about rewards. EMERGENCY ONLY.

*function* emergencyWithdraw() public **{**

PoolInfo storage pool **=** poolInfo[*0*];

UserInfo storage user **=** userInfo[**msg.sender**];

pool.lpToken.safeTransfer(address(**msg.sender**), user.amount);

**emit** EmergencyWithdraw(**msg.sender**, user.amount);

user.amount **=** *0*;

user.rewardDebt **=** *0*;

**}**

All fund is withdrawn and the rewardDebt state is cleared so that next time the user stake he will pretty much be treated as a new user. For the rest of the users, the reward calculation will be as if the fund is removed right after last update. The only side effect of the call is the pending reward amount of this user is lost. Which we do not really care that much. The unpaid reward will stay in the reward pool.

One can see that only token transfer and setting values are involved in the function. NO update logic or reward calculation logic is called. This will pretty much eliminate all the risks to fail the tx.

## Key API

User:

*function* deposit() public payable

*function* withdraw(uint256 \_amount) public

*function* emergencyWithdraw() public

*function* setUpstreamURL(string memory \_url) public

Admin:

(Or maybe we do not need admin, let the owner serve the admin role)

*function* setAdmin(address \_adminAddress) public onlyOwner

*function* addToAllowList(address \_allowlistAddress) public onlyAdmin

*function* removeFromAllowList(address \_allowlistAddress) public onlyAdmin

*function* getAllowList(uint256 index) external view returns (address)

*function* getAllowListCount() external view returns (uint256)

Proxy:

*function* getAmount(address \_user) external view returns (uint256)

*function* getUpstreamURL (address \_user) external view returns (string memory)

(The stored info can be a JSON containing more info or an address to some server store extra info)

Allow accept token:

**receive()** external payable

Note: most internal data should be exposed so that UI can display necessary info.

Make sure the exposed API is friendly for MultiCall (<https://github.com/makerdao/multicall>) to use.

Events

*event* Deposit(address indexed user, uint256 amount);

*event* Withdraw(address indexed user, uint256 amount);

*event* EmergencyWithdraw(address indexed user, uint256 amount);

*event* AddToAllowList(address indexed user);

*event* RemoveFromAllowList(address indexed user);