Modified ERC20 Token

(A) The Project Goal & Description

The goal of the project is to further exercise gained knowledge from programming of smart contracts in Solidity as well as decentralized applications (DAPPS) in javascript. See lectures 6, 7, and demo-exercise in Moodle, which is a prerequisite. The task is to enrich the functionality of standardized ERC20 token by:

1. **Capping the total supply** of tokens to a value specified in the constructor, which means that tokens in circulation cannot surpass the maximum value of the cap. You can start with the total supply 0.
2. **Minting functionality** that allows anyone with the *mintingAdmin* role to mint a maximum number of tokens TMAX within a day to any address (with verified identity – see later). Minting of higher amounts of tokens than TMAX can be made by a majority consensus of users with the *mintingAdmin* role (i.e., threshold-based multisig). The global value of TMAX can be changed by the majority consensus of *mintingAdmin* role. The minting should be possible to make on a single address as well as batches of addresses, requiring majority consensus of the *mintingAdmin* role for amounts higher than TMAX.
3. **Simple identity verification** of all token holders, which will enable to receive the tokens only to user addresses that have been verified by any centralized identity provider (IDP) from the list of identity providers represented by their addresses. The list of identity providers will be managed by *idpAdmin* role based on the majority consensus (alike in the previous case). (Note that *idpAdmin* role != list of identity providers). Already verified users can be revoked by the majority consensus of *idpAdmin* role, while revoked users can be later reactivated by the majority consensus of the same role. The user should also have default expiration UEXP, after which they are no longer considered verified; on the other hand they can be renewed by the majority consensus of *idpAdmin* role. Note that users can submit verifiable credentials signed by the 3rd party identity providers who do not have access to the blockchain platform and are not any users here (see also recommendations below).
4. **Disallow the users to burn tokens** by sending them to 0x0 address.
5. **The management of various admin roles**. On top of enabling the approvals of actions by a majority consensus of a certain role for the purposes above (i.e., minting, **C~~R~~U~~D~~** on identity providers), the set of admins can be modified by a majority consensus by themselves. In particular, the new member can be added and deleted.
6. **Transfer restrictions on the users**. Even thought the token holders (i.e., users) need to have verified identities to possess tokens, we will make certain transfer restrictions on them that will allow the users to transfer only a limited amount of tokens per day (this is just an example while in practice there might be different use cases). For this purpose, transfer restriction pattern should be used while the maximum amount for transfer should be set to the default guaranteed value TRANSFERLIMIT (i.e., constant) during the deployment of

contracts for all users, while the particular users might have certain exceptions (i.e., above TRANSFERLIMIT) that are managed by the majority consensus of the role of *restrAdmin*.

The next part of the project is to make DAPP that will run in the client browser and enable to interact with the smart contracts made. DAPP can use any SW wallet, such as Metamask to confirm actions by users. DAPP should be capable to:

1. Display the balance of the user and show forms for transferring the tokens directly or through a delegation (see the default functionality of ERC20).
2. Display the addresses belonging to particular roles.
3. Mint admins should know how much tokens they have already minted today and how much they can still mint.
4. Display all the roles that the user have and according to them display forms for particular actions.

It is sufficient that your project will work on a local blockchain emulated by ganache-cli.

(B) Background

To understand some basic background for development of smart contracts, see the demonstration exercise at https://moodle.vut.cz/mod/folder/view.php?id=307294&forceview=1. The exercise contains detailed guide for development and testing of smart contracts. It is highly recommended to pass this demo before starting to work on the current project. Note that this demo emulates DAPPs functionality by the unit tests that call smart contract methods.

(C) Recommendations

- Maximize the utilization of contracts made by OpenZeppelin – https://github.com/OpenZeppelin/openzeppelin-contracts. See mainly ERC20 contract as well as EnumerableSet that might be useful for some purposes.
- You can utilize external contracts from your extended ERC20 contract for delegation of some operations, such as identity verification and transfer restrictions. Since external contract are created within the parent contract (i.e., extended ERC20), the owner of this contract is always the parent contract, which you might leverage in some cases (see also Ownable of OpenZeppelin).
- Think of universal approach to roles and the actions they made, which can be represented by a certain contract (or two contracts) with multiple instances – each role using a single instance.
- Since Solidity is the OOP language, you can leverage its features. For example to extend the ERC20 contracts (with the cap) by minting capability into MintableERC20, which can be later extended to IDManagedMintableERC20 that will support the identity verification by external

contract meeting a certain interface *IIdentityRegistry.verify(address recipient) bool.* And so on.

- You should emit events for all actions that are made using smart contracts as a means to inform the users sitting at DAPPs who are listening to those events.
- For time-based checks, use the variables of environment, such as block.timestamp.
- You might need to "overload" and extend some existing methods of ERC20 contract by your methods to accomplish the expected funtionality, e.g., for minting, etc. However, no modification of parent ERC20 contract is allowed.
- Identity management contract can add verified users to its registry upon providing some IDP's the signature on a certain message such as "*The user with address {XY} has verified identity.*", representing the abstracted verifiable credentials of the user since the user is the holder of the private key associated with the address XY. Therefore, the user can obtain signed verifiable credentials from the identity provider anytime and later submit it to the identity management smart contract. Hence, the signature verification is not the part of the native signature verification of EVM by msg.sender. Instead, ecrecover should be used. Note that IDP is can be highly abstracted and should enable only signing of verifiable credentials of users – e.g., you can emulate it in javascript upon user's request.
- Use at least version 0.8.13 of Solidity. https://docs.soliditylang.org/en/v0.8.13/

(D) Documentation

You should describe your solution and comment on the features it has, what are the disadvantages inherent to the instructions or you particular solution and how it can be improved. Next, report on gas measurements of particular operations that your solution supports. The easies way to obtain the gas measurements are javascript tests as well as logs of local blockchain run by ganache-cli.

(E) What to Hand In?

Compress all the files into zip archive. The submitted files should contain:

1) All project files except dependencies in *node_modules* and *compiled binaries* of smart contract files. Dependencies should be installed automatically after issuing *npm install.*

2) Readme file.

3) Documentation in pdf.