https://etherscan.io/token/0xb46eda6219ba121ce9280388e7afb7dc84be3ff2#code

Medium severity issues

• Initial Token Distribution / Centralization

In the constructor function, the deployer is minting tokens for his wallet address. All the initial supply are sent to the contract deployer when deploying the contract. This could be a centralization risk, the deployer can distribute those tokens without obtaining the consensus of the community.

```
167 constructor() public payable{
168 require(msg.value > 0.3 ether);
169 address(0x06806458405C55E40D75Bd0fE1732500Cd1C229c).transfer(msg.value);
170 name = "TheYouthPay";
171 symbol = "TYP";
172 decimals = 0;
173 initialSupply = 5000000000;
174 totalSupply_ = 5000000000;
175 balances[owner] = totalSupply_;
176 emit Transfer(address(0), owner, totalSupply_);
177 }
```

✓ Recommendation

We recommend the team to be transparent regarding the initial token distribution process, and the team shall make enough efforts to restrict the access of the private key, We also advise the client to adopt Multisig, Timelock, and/or DAO in the project to manage the specific account in this case.

• Owner Role / Centralized Risk

Only the owner role has authority over the functions shown below. Any compromise to the _owner account may allow the hacker to take advantage of this authority.

- ✓ freezeAccount
- ✓ unfreezeAccount

✓ Recommendation

https://etherscan.io/token/0xb46eda6219ba121ce9280388e7afb7dc84be3ff2#code

The risk describes the current project design and potentially makes iterations to improve the security operation and level of decentralization, which in most cases cannot be resolved entirely at the present stage. We advise the client to carefully manage the privileged account's private key to avoid any potential risks of being hacked. In general, we strongly recommend centralized privileges or roles in the protocol be improved via a decentralized mechanism or smart-contract-based accounts with enhanced security practices, e.g., multi-signature wallets.

Low severity issues

• Unlocked Pragma Used

Contracts should be deployed with the same compiler version and flags that they have been tested the most with. Locking the pragma helps ensure that contracts do not accidentally get deployed using, for example, the latest compiler which may have higher risks of undiscovered bugs. Contracts may also be deployed by others and the pragma indicates the compiler version intended by the original authors.

```
// bad
pragma solidity ^0.4.4;

// good
pragma solidity 0.4.4;
```

✓ Recommendation

Should lock pragmas to a specific compiler version.

• Improper Usage of public and external Type

public functions that are never called by the contract could be declared as external. external functions are more efficient than public functions.

- ✓ Burn
- ✓ unfreezeAccount
- ✓ freezeAccount
- ✓ transferFrom
- ✓ transfer

✓ Recommendation

https://etherscan.io/token/0xb46eda6219ba121ce9280388e7afb7dc84be3ff2#code

Consider using the external attribute for public functions that are never called within the contract.