Crypto Stock Foundation

Smart Contracts Audit

v1.0
Vittorio Minacori
BC1 - Blockchain Pioneers

Nov 25, 2020

Disclaimer

The audit makes no statements or warranties about utility of the code, safety of the code, suitability of the business model, regulatory regime for the business model, or any other statements about fitness of the contracts to purpose, or their bugfree status. The audit documentation is for discussion purposes only.

Overview

The audit was performed by **BC1 - Blockchain Pioneers** (https://www.bc1.tech) on all the Smart Contracts provided by the **Crypto Stock Foundation** team.

This report can be found here CryptoStockFoundation-BC1-Audit-Report-V1.0

NOTE:
Code to analyze was provided in a flattened file, without any information regarding tests, version of development tools or libraries which have been used, compiler version, implementation logic.

Provided source code can be found here src/GoogleConstr_Flat.sol

To audit the code, it was required by **BC1 - Blockchain Pioneers** team, to reorganize everything in a standard structure, to lock tools and libraries version, to build tests.

Reorganized code can be found here.

Audited smart contracts are in the ***contracts*** folder.

Built tests are in the ***test*** folder.

Assuming smart contracts will be compiled using ***solc v0.7.4+commit.3f05b770.Emscripten.clang***.

Assuming smart contracts use the ***@openzeppelin/contracts*** library at version **3.2.2-solc-0.7**.

Brief

Audit was performed on **AcsERC20_Google** which extends **AcsERC20**.
We deduct that the last will be used as a "template" to generate other tokens with the same behaviours.

AcsERC20_Google has only a *constructor* where *name*, *symbol*, *initialSupply* and *decimal* are hard coded.
It supposes that **333.630.000.000 xGOOG** will be generated and that token has 6 decimals so, basically, it will generate 333630000000000000 token units.

AcsERC20 itself seems inspired by the preset token distributed by OpenZeppelin (ERC20PresetMinterPauser) with some changes in logic, also if comments have not been updated from the original one (see below).

Token is Mintable, Burnable, Pausable only for addresses with the defined role.
Token has not a max supply.
Token has both an owner and an access control behaviour.

Everything will be discussed in detail in the code analysis below.

Best Practices

Blockchain developers following best practices, make audits more meaningful, by allowing efforts to be focused on subtle and project-specific issues rather than the fulfillment of general guidelines. Avoiding code duplication is a good example of a good engineering practice which increases the potential of any security audit. We consider a list of few points that should be enforced in any good project that aims to be deployed on the Ethereum blockchain.

Hard Requirements

❌ The code is provided as a Git repository to allow the review of future code changes**.**

❌ Code duplication is minimal, or justified and documented.

⚠️ Libraries are properly referred to as package dependencies, including the specific version(s).

✅ The code compiles with the latest Solidity compiler version.

⚠️ There are no compiler warnings, or warnings are documented.

❌ There are tests.

❌ The test coverage is available or can be obtained easily.

Soft Requirements

❌ The code is well documented.

⚠️ Functions are grouped together according either to the Solidity guidelines, or to their functionality.

❌ Use of *modifiers* to avoid code duplication.

✅ Use of *SafeMath* library to prevent integer overflow (unnecessary since solidity 0.8.x).

Audit Details

Arithmetic Safety Audit ✅

The arithmetic security audit is divided into three parts: integer overflow audit, integer underflow audit and operation precision audit.

Integer Overflow Audit ✅

Solidity can handle 256 bits of data at most. When the maximum number increases, it will overflow. If the integer overflow occurs in the transfer logic, it will make the amount of transfer funds miscalculated, resulting in serious capital risk.

Integer Underflow Audit ✅

Solidity can handle 256 bits of data at most. When the minimum number decreases, it will underflow. If the integer underflow occurs in the transfer logic, it will make the amount of transfer funds miscalculated and lead to serious capital risk.

Operation Precision Audit ✅

Solidity performs type coercion in the process of multiplication and division. If the precision risk is included in the operation of capital variable, it will lead to user transfer logic error and capital loss.

Audit Result: Passed
Security Recommendation: No.

Competitive Competition Audit ✅

The competitive competition audit is divided into two parts: reentrancy audit and transaction ordering dependence audit. With competitive vulnerabilities, an attacker can modify the output of a program by adjusting the execution process of transactions with a certain probability.

Reentrancy Audit ✅

Reentrancy occurs when external contract calls are allowed to make new calls to the calling contract before the initial execution is complete. For a function, this means that the contract state may change in the middle of its execution as a result of a call to an untrusted contract or the use of a low level function with an external address.

Transaction Ordering Dependence Audit ✅

Since miners always get rewarded via gas fees for running code on behalf of externally owned addresses (EOA), users can specify higher fees to have their transactions mined more quickly. Since the Ethereum blockchain is public, everyone can see the contents of others' pending transactions. This means if a given user is revealing the solution to a puzzle or other valuable secret, a malicious user can steal the solution and copy their transaction with higher fees to preempt the original solution. If developers of smart contracts are not careful, this situation can lead to practical and devastating front-running attacks.

Audit Result: Passed
Security Recommendation: No.

Access Control Audit ⚠️

Access control audit is divided into two parts: privilege vulnerability audit and overprivileged audit.

Privilege Vulnerability Audit ✅

Smart contracts with privilege vulnerability, attackers can weigh their own accounts to gain higher execution privileges.

Overprivileged Audit ⚠️

Overprivileged auditing focuses on whether there are special user privileges in audit contracts, such as allowing a user to unlimitedly mine tokens.

Audit Result: Warning
Security Recommendation: Check the Audit Results / Low severity section for details.

Security Design Audit ⚠️

Security design audit is divided into five parts: security module usage, compiler version security, hard-coded address security, sensitive function usage security and function return value security.

Security Module Usage ✅
The security module usage audit whether the smart contract uses the SafeMath library function to avoid overflow vulnerabilities; if it does not, whether the transfer amount is strictly checked during the execution.

Compiler Version Security Audit ⚠️
Compiler version security focuses on whether the smart contract explicitly indicates the compiler version and whether the compiler version used is too low to throw an exception.

Hard Coded Address Security Audit ✅
Hard-coded address security audit static addressed in the smart contract to check whether there is an exception to the external contract, thus affecting the execution of this contract.

Sensitive Functions Audit ✅
Sensitive functions audit checks whether the smart contract uses the not recommended functions such as fallback, call and tx.origin.

Function Return Value Audit ⚠️
Function return value audit mainly analyzes whether the function correctly throws an exception, correctly returns to the state of the transaction.

Audit Result: Warning
Security Recommendation: Check the Audit Results / Low severity section for details.

Denial of Service Audit ✅
Denial of service attack sometimes can put the smart contract offline forever by maliciously behaving when being the recipient of a transaction, artificially increasing the gas necessary to compute a function, abusing access controls to access private components of smart contracts, taking advantage of mixups and negligence and so on.

Audit Result: Passed
Security Recommendation: No.

Gas Optimization Audit ✅
If the computation of a function in a smart contract is too complex, such as the batch transfer to a variable-length array through a loop, it is very easy to cause the gas fee beyond the block's gas Limit resulting in transaction execution failure.

Audit Result: Passed
Security Recommendation: No.

Design Logic Audit ✅
In addition to vulnerabilities, there are logic problems in the process of code implementation, resulting in abnormal execution results.

Audit Result: Passed
Security Recommendation: No.

Audit Results

Critical severity ✅

No critical severity issues were found.

High severity ✅

No high severity issues were found.

Medium severity ✅

No medium severity issues were found.

Low severity ⚠️

- **Function Return Value Audit:** *transfer()* and *transferFrom()* use **SafeMath** functions of the extended OpenZeppelin library, which will cause them to throw instead of returning false. Not a security issue but departs from standard. It is inherited from OpenZeppelin library and it is a very low severity issue.
- **Overprivileged Audit: AcsERC20** has no way to define the maximum number of tokens mintable. If there is a maximum number of tokens to be ever minted we would suggest to set a max cap of mintable tokens or add a *finishMinting* function to disable minting. Since there is no finish minting method, minters could generate more tokens than declared. We mark this as a low severity issue.
- **Compiler Version Security Audit:** Compiler version is not provided. The defined *pragma* allows ^0.7.0. Since contracts compile with the latest solidity version we assume it will be 0.7.4 so we mark this as a very low severity issue.

Note: *approve()* is affected by the issue related on ERC20 standard approve. To prevent attack vectors like the one described here and discussed here, clients SHOULD make sure to create user interfaces in such a way that they set the allowance first to 0 before setting it to another value for the same spender.
We mark this as a very low severity issue.

Testing Process

The **BC1 - Blockchain Pioneers** team reorganized the source code provided and added a standard environment to compile, tests and analyze contracts.

Fixing package version

Source code here https://github.com/bc1tech/crypto-stock-foundation-audit.

Packages:

- **truffle**: 5.1.49
- **hardhat**: 2.0.3
- **solc**: 0.7.4
- **@openzeppelin/contracts**: 3.2.2-solc-0.7

Code build can be found here https://travis-ci.com/github/bc1tech/crypto-stock-foundation-audit.

Lint failed ❌
https://travis-ci.com/github/bc1tech/crypto-stock-foundation-audit/jobs/447411195

```
contracts/GoogleConstructor.sol
  7:1  warning  Contract name must be in CamelCase  contract-name-camelcase
✖ 1 problem (0 errors, 1 warning)
```

Test passed ✅
Using Truffle https://travis-ci.com/github/bc1tech/crypto-stock-foundation-audit/jobs/447411191
Using Hardhat https://travis-ci.com/github/bc1tech/crypto-stock-foundation-audit/jobs/447411192

```
  94 passing (8s)
```

Test Coverage 98% ✅
https://coveralls.io/github/bc1tech/crypto-stock-foundation-audit
11 of 12 branches covered (91.67%).
30 of 30 relevant lines covered (100.0%).
26.63 hits per line.

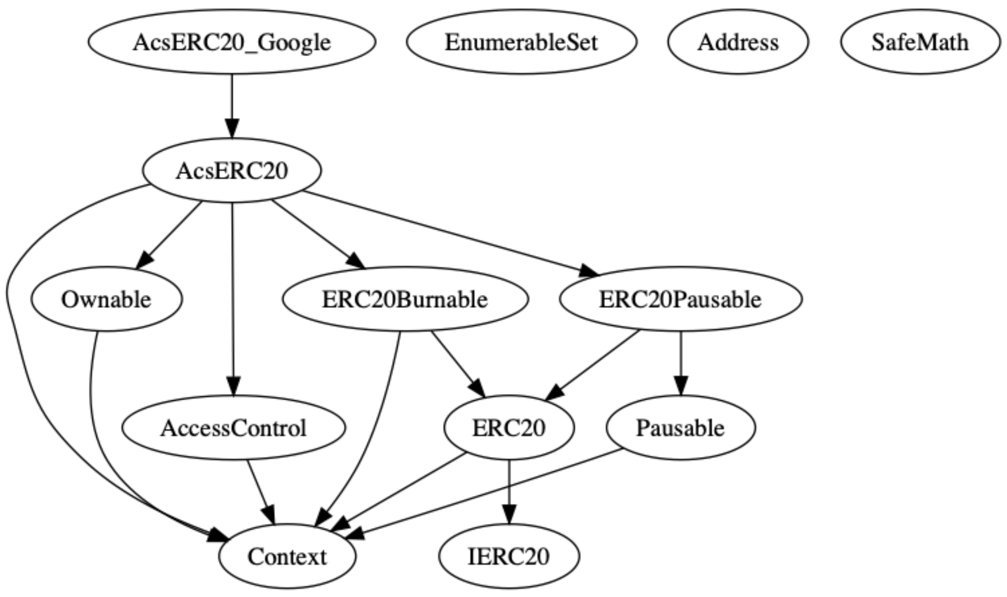
Impossible to cover this line.

Structure Analysis

Inheritance Tree Graph

The **BC1 - Blockchain Pioneers** team built an inheritance tree graph for the smart contract provided.

Graphs can be found at analysis/inheritance-tree/AcsERC20_Google.png.



Control Flow Graph

The **BC1 - Blockchain Pioneers** team built a control flow graph for the smart contract provided.

Graphs can be found at analysis/control-flow/AcsERC20_Google.png.

File Description Table

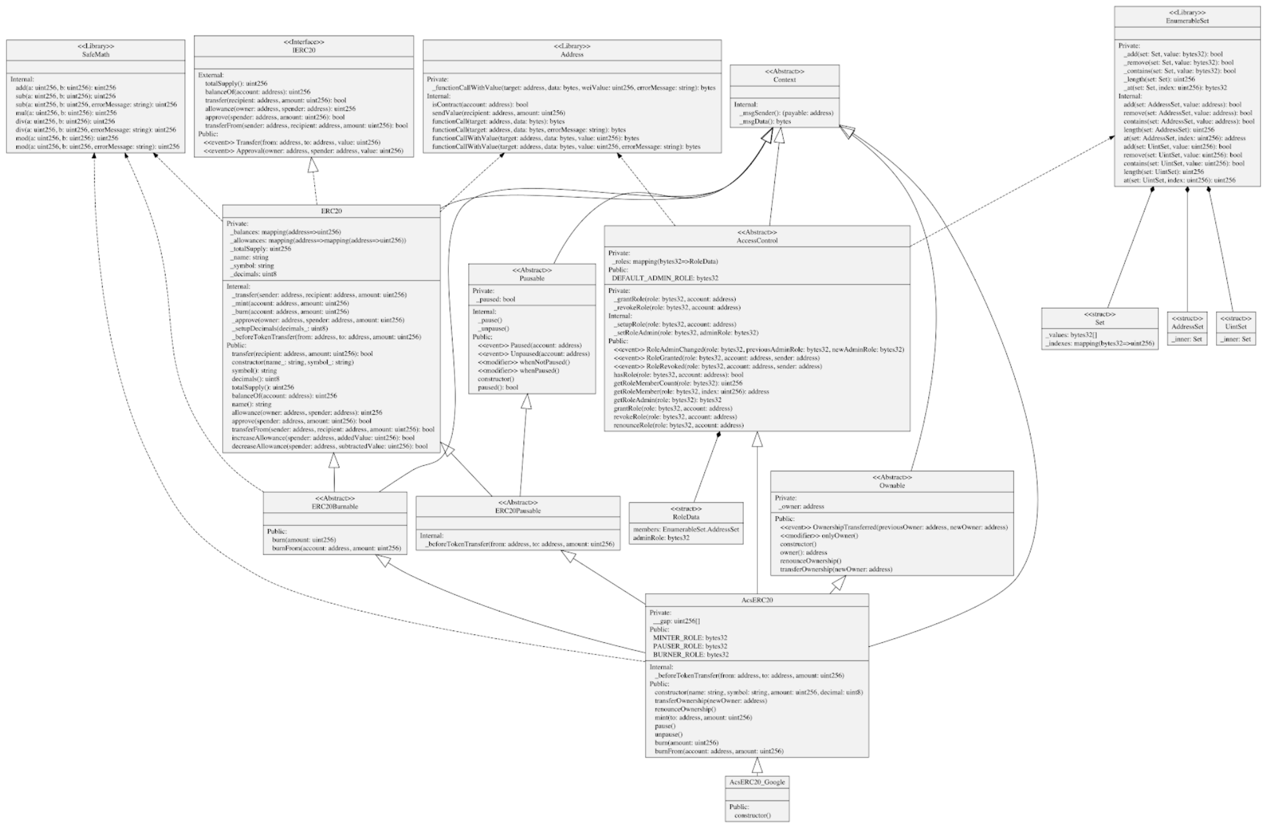
The **BC1 - Blockchain Pioneer**s team built a description table for the smart contract provided.

Tables can be found at analysis/description-table/AcsERC20_Google.md.

UML

The **BC1 - Blockchain Pioneer**s team built a UML for the smart contract provided.

UML can be found at analysis/uml/AcsERC20_Google.svg.



Code Analysis

AcsERC20_Google

Source code: [contracts/GoogleConstructor.sol]()

This contract is the main token contract.
AcsERC20_Google extends **AcsERC20**.

```
contract AcsERC20_Google is AcsERC20
```

Constructor

```
constructor() AcsERC20("Alphabet Inc.", "xGOOG", 333630000000, 6) {}
```

The contract constructor initiates the **AcsERC20** with token *name*, *symbol*, *amount* and *decimal*.

Suggestion:

- ⚠ Use "e" to better write numbers to avoid typos (like 333630e6).
- ⚠ As ETH is based on 18 decimals, tokens usually opt for a value of 18, imitating the relationship between Ether and Wei. In case it will be required to exchange tokens for ETH in a DEX or using a Smart Contract, it will be really complex to set up an exchange rate based on different numbers of units. If it is not required by your logic use 18 instead.

AcsERC20

Source code: [contracts/lib/ERC20AcsConstructor.sol]()

This contract contains logic implementation.
AcsERC20 extends **Context, AccessControl, Ownable, ERC20Burnable, ERC20Pausable**.

```
contract AcsERC20 is Context, AccessControl, Ownable, ERC20Burnable, ERC20Pausable
```

As said above this contract is "inspired" by the *ERC20PresetMinterPauser* preset token, distributed by OpenZeppelin from where they are also comments copy/pasted in class definition.
Anyway they are not all related to the below code.
For instance it says *"ability for holders to burn (destroy) their tokens"*, but they really don't because of only addresses with the BURNER role will be able to do that.

```
/**
* @dev {ERC20} token, including:
*
*  - ability for holders to burn (destroy) their tokens
*  - a minter role that allows for token minting (creation)
*  - a pauser role that allows to stop all token transfers
*
* This contract uses {AccessControl} to lock permissioned functions using the
* different roles - head to its documentation for details.
*
* The account that deploys the contract will be granted the minter and pauser
* roles, as well as the default admin role, which will let it grant both minter
* and pauser roles to other accounts.
*/
```

Required:

- ❌ Remove pasted comments or edit them in order to better describe the token behaviours. Reading comments people may expect something that token, really, doesn't allow.

Constructor

```
/**
* @dev Grants `DEFAULT_ADMIN_ROLE`, `MINTER_ROLE` and `PAUSER_ROLE` to the
* account that deploys the contract.
*
* See {ERC20-constructor}.
*/

constructor(string memory name, string memory symbol, uint256 amount, uint8 decimal)
ERC20(name, symbol) {
   _setupRole(DEFAULT_ADMIN_ROLE, _msgSender());
   _setupRole(MINTER_ROLE, _msgSender());
   _setupRole(PAUSER_ROLE, _msgSender());
   _setupRole(BURNER_ROLE, _msgSender());
   _setupDecimals(decimal);
   //Mint amount to the sender
   _mint(_msgSender(), amount * (10 ** uint256(decimals())));
}
```

The contract constructor initiates the **ERC20** with token *name* and *symbol*. It uses *amount* and *decimal* to create an initial supply and send it to *msg.sender*. Again comments are copy/pasted from *ERC20PresetMinterPauser* and they say that DEFAULT_ADMIN_ROLE, MINTER_ROLE and PAUSER_ROLE will be granted to contract deployer and say nothing about the BURNER_ROLE.
Due that the token has no max supply, the amount params seems to be unuseful here. It could be removed

from the token constructor and tokens can be minted in a separate transaction. If a token should be created using the AcsERC20 template and it requires not to have an initial amount, zero must be used as amount.
It means that the *_mint* function will be used like **_mint(_msgSender(), 0 * (10 ** uint256(decimals())));** causing the minting and transferring of 0 tokens.

Suggestion:

- ⚠ Rename *decimal* parameter to *decimals* for consistency.
- ⚠ Remove *amount* as parameter and mint token in a following transaction.

  If you want to keep amount as a parameter:

  - ⚠ Rename amount to initialAmount to better describe that it is only an initial amount and not a cap or max supply.
  - ⚠ By renaming *decimal* to *decimals*, the naming issue is that it can't be used decimals() in the *_mint* function due the **TypeError: Type is not callable**.
    So use the new *decimals* parameter instead of the decimals function like this
    _mint(_msgSender(), amount * (10 ** uint256(decimals)));

Required:

- ❌ Remove pasted comments or edit them in order to better describe the token behaviours. Reading comments people may expect something that token, really, doesn't allow.

Modifiers

There is no use of modifiers. Some code is repeated in *pause*/*unpause*, *burn*/*burnFrom*. In order to avoid editing some access logic and forgetting to update each occurrence it must be a best practice to define a function modifier and use it in methods definition.

For instance we could define a *onlyPauser* modifier and use it in *pause* method like the below example.

onlyPauser

```
modifier onlyPauser() {
    require(hasRole(PAUSER_ROLE, _msgSender()), "AcsERC20: PAUSER role required");
    _;
}

function pause() public virtual onlyPauser {
    _pause();
}

function unpause() public virtual onlyPauser {
    _unpause();
}
```

Suggestion:

- ⚠ Use modifiers to define MINTER, BURNER, PAUSER role behaviours.

Public Methods

transferOwnership

```
/**
* @dev Transfers ownership of the contract to a new account (`newOwner`).
* Can only be called by the current owner.
* Revoke all roles to the previous owner
* Grant all roles to the new owner
*/
function transferOwnership(address newOwner) public override {
   //addStakeholder(newOwner);
   super.transferOwnership(newOwner);
   //need to have almost DEFAULT_ADMIN_ROLE role
   _setupRole(DEFAULT_ADMIN_ROLE, _msgSender());
   grantRole(DEFAULT_ADMIN_ROLE, newOwner);
   grantRole(MINTER_ROLE, newOwner);
   grantRole(PAUSER_ROLE, newOwner);
   grantRole(BURNER_ROLE, newOwner);

   revokeRole(MINTER_ROLE, _msgSender());
   revokeRole(PAUSER_ROLE, _msgSender());
   revokeRole(BURNER_ROLE, _msgSender());
   revokeRole(DEFAULT_ADMIN_ROLE, _msgSender());
}
```

The *transferOwnership* method is overriding the parent method from *Ownable*.
It is intended to transfer contract ownership to a new owner with all the roles attached. It seems that the *onlyOwner* modifier is unused in all the code sources so there could be no reason to have a token owner. The contract, infact, extends the *AccessControl* contract provided by OpenZeppelin that allows users to manage roles. In the top of smart contract there are 3 roles defined (MINTER_ROLE, PAUSER_ROLE, BURNER_ROLE) other than the DEFAULT_ADMIN_ROLE. Having an owner seems to be an unnecessary behaviour because he can do the same actions a DEFAULT_ADMIN_ROLE can.
Because of this we imagine that it has been necessary to call **_setupRole(DEFAULT_ADMIN_ROLE, _msgSender());** to be able to grant and revoke roles in cases where admin was changed in time. But *_setupRole* is intended to be called only during contract construction as expressed in the AccessControl contract below. It would be better to remove it in transferOwnership.

```
// File: @openzeppelin/contracts/access/AccessControl.sol

/**
* @dev Grants `role` to `account`.
*
* If `account` had not been already granted `role`, emits a {RoleGranted}
* event. Note that unlike {grantRole}, this function doesn't perform any
* checks on the calling account.
*
* [WARNING]
* ====
* This function should only be called from the constructor when setting
* up the initial roles for the system.
*
* Using this function in any other way is effectively circumventing the admin
* system imposed by {AccessControl}.
* ====
*/
function _setupRole(bytes32 role, address account) internal virtual {
    _grantRole(role, account);
}
```

Suggestion:

- ✅ Remove unnecessary comment *//addStakeholder(newOwner);*
- ⚠️ Remove inheritance from *Ownable* and use *AccessControl* with modifiers as described above. Move *transferOwnership* to a custom method (i.e. *transferAdmin*) using a *onlyAdmin* modifier. Inside this method grants roles to new admin and revokes roles from the old one as already done.

Required:

- ❌ Do not use *_setupRole* outside the contract constructor.

renounceOwnership

```
/**
* @dev Leaves the contract without owner. It will not be possible to call
* `onlyOwner` functions anymore. Can only be called by the current owner.
*
* NOTE: Renouncing ownership will leave the contract without an owner,
* thereby removing any functionality that is only available to the owner.
*/
function renounceOwnership() public view override {
    require(owner() == address(0), "AcsERC20: renounceOwnership function was disabled;
please use transferOwnership");
}
```

The *renounceOwnership* method is overriding the parent method from *Ownable*.
As described in *transferOwnership* it could be unnecessary to have an owner by having an admin. So this method could be removed and replaced to an override of *renounceRole* or *revokeRole* to be sure to not to leave a contract without an admin. Having the owner just for convenience should be replaced by a better Access Control handling.
Anyway this method is requiring that **owner() == address(0)** that could be impossible by analyzing the code. This assertion will always be false because its purpose is to disable renouncing ownership.
It can be replaced with a *revert* like this
revert("AcsERC20: renounceOwnership function was disabled; please use transferOwnership");

Suggestion:

- ⚠️ Remove inheritance from *Ownable* and use *AccessControl* with modifiers as described above. Override AccessControl methods if you want to avoid leaving a contract without admin.

Required:

- ❌ Use *revert* instead of an always failing *require*.

mint

```
/**
* @dev Creates `amount` new tokens for `to`.
*
* See {ERC20-_mint}.
*
* Requirements:
*
* - the caller must have the `MINTER_ROLE`.
*/
function mint(address to, uint256 amount) public virtual {
   require(hasRole(MINTER_ROLE, _msgSender()), "ERC20PresetMinterPauser: must have
minter role to mint");
   _mint(to, amount);
}
```

A method to mint new tokens. Only addresses with MINTER role can generate new tokens. There is no cap in token supply so minters could generate as tokens as they want.

Suggestion:

- ⚠️ Use a modifier instead of having a require.

pause/unpause

```
/**
* @dev Pauses all token transfers.
*
* See {ERC20Pausable} and {Pausable-_pause}.
*
* Requirements:
*
* - the caller must have the `PAUSER_ROLE`.
*/
function pause() public virtual {
   require(hasRole(PAUSER_ROLE, _msgSender()), "ERC20PresetMinterPauser: must have
pauser role to pause");
    _pause();
}

/**
* @dev Unpauses all token transfers.
*
* See {ERC20Pausable} and {Pausable-_unpause}.
*
* Requirements:
*
* - the caller must have the `PAUSER_ROLE`.
*/
function unpause() public virtual {
    require(hasRole(PAUSER_ROLE, _msgSender()), "ERC20PresetMinterPauser: must have
pauser role to unpause");
    _unpause();
}
```

These two methods allow to pause or unpause token transfers. Read considerations below.

Suggestion:

- ⚠ Use a modifier instead of having a require.

burn/burnFrom

```
/**
* @dev Destroys `amount` tokens from the caller.
*
* See {ERC20-_burn}.
*
* Added a BURNER_ROLE conditions
*/
function burn(uint256 amount) public override(ERC20Burnable) {
    require(hasRole(BURNER_ROLE, _msgSender()), "AcsERC20: must have burner role to
burn");
    _burn(_msgSender(), amount);
}

/**
* @dev Destroys `amount` tokens from `account`, deducting from the caller's
* allowance.
*
* See {ERC20-_burn} and {ERC20-allowance}.
*
* Requirements:
```

```
*
* - the caller must have allowance for ``accounts``'s tokens of at least
* `amount`.
*
* Added a BURNER_ROLE conditions
*/
function burnFrom(address account, uint256 amount) public override(ERC20Burnable) {
   require(hasRole(BURNER_ROLE, _msgSender()), "AcsERC20: must have burner role to
burn");
   uint256 decreasedAllowance = allowance(account, _msgSender()).sub(amount, "ERC20:
burn amount exceeds allowance");

   _approve(account, _msgSender(), decreasedAllowance);
   _burn(account, amount);
}
```

These two methods allow to burn tokens from its own address or from an address which has previously allowed using the standard ERC20 function *approve*. Usually each address is able to burn its tokens (or approved tokens by another address), but in this case also a BURNER role is required.
Anyway the code after role requirement is a copy/paste of the inherited *ERC20Burnable* contract. These lines could be replaced by a *super* call, otherwise, by analyzing the code, it is not necessary to inherit the ERC20Burnable contract because its methods are never called.

Suggestion:

- ⚠️ Use a modifier instead of having a require.

Required:

- ❌ User *super* to call parent method and remove duplicate code or remove ERC20Burnable inheritance.

Internal Methods

_beforeTokenTransfer

```
function _beforeTokenTransfer(address from, address to, uint256 amount) internal
virtual override(ERC20, ERC20Pausable) {
   super._beforeTokenTransfer(from, to, amount);
   //addStakeholder(to);
}
```

Necessary because the derived contract must implement *_beforeTokenTransfer* from ERC20 and ERC20Pausable.

Suggestion:

- ✅ Remove unnecessary comment *//addStakeholder(to);*

Private Property

__gap

```
uint256[50] private __gap;
```

Unused.

Required:

- ❌ Remove the unused property.

Style Guide

Order of Layout

Layout contract elements in the following order:

1. Pragma statements
2. Import statements
3. Interfaces
4. Libraries
5. Contracts

Inside each contract, library or interface, use the following order:

1. Type declarations
2. State variables
3. Events
4. Functions

Suggestion:

- ✅ Nothing.

Order of Functions

Ordering helps readers identify which functions they can call and to find the constructor and fallback definitions easier.

Functions should be grouped according to their visibility and ordered:

1. constructor
2. receive function (if exists)
3. fallback function (if exists)
4. external
5. public
6. internal
7. private

Suggestion:

- ⚠️ Move *_beforeTokenTransfer* as last in the code.

Function Declaration

For short function declarations, it is recommended for the opening brace of the function body to be kept on the same line as the function declaration.
The closing brace should be at the same indentation level as the function declaration.
The opening brace should be preceded by a single space.

The modifier order for a function should be:

1. Visibility
2. Mutability
3. Virtual
4. Override
5. Custom modifiers

For constructor functions on inherited contracts whose bases require arguments, it is recommended to drop the base constructors onto new lines in the same manner as modifiers if the function declaration is long or hard to read.

Suggestion:

- ✅ Drop *ERC20(name, symbol)* onto new line in constructor.

Naming Convention

Naming conventions are powerful when adopted and used broadly. The use of different conventions can convey significant meta information that would otherwise not be immediately available.
The naming recommendations given here are intended to improve the readability, and thus they are not rules, but rather guidelines to try and help convey the most information through the names of things.

Contract and Library Names:

1. Contracts and libraries should be named using the CapWords style. Examples: SimpleToken, SmartBank, CertificateHashRepository, Player, Congress, Owned.
2. Contract and library names should also match their filenames.
3. If a contract file includes multiple contracts and/or libraries, then the filename should match the core contract. This is not recommended however if it can be avoided.

Suggestion:

- ⚠️ Rename *AcsERC20_Google* in CamelCase.
- ⚠️ Rename *contracts/GoogleConstructor.sol* and *contracts/lib/ERC20AcsConstructor.sol* as the same name of the contract.

Considerations

The **BC1 - Blockchain Pioneers** team haven't received tests about code nor a detailed description of the entire project. We reserve to edit the below considerations about token behaviours if more details will be available later.

⚠️ **Not capped token:** there is no upper cap to token supply. It means that token owner/minter could generate more tokens than declared or backed by real assets. So if there is some logic in token issuance it should be coded into the mint function otherwise owner/minter are able to do whatever they want. Holders or investors must be careful and they must rely on centralized token governance.

⚠️ **Pausable behaviour:** there could be many reasons to have this feature in an ERC20 token but a malicious PAUSER could try to set the token as *paused* and do not never unpause it. If for some reasons nobody will be able to access to the admin/owner wallet, or some bug will be found in AccessControl or Ownable OpenZeppelin contracts or in the previous mentioned transferOwnership, renounceOwnership, etc., token could remain in a stuck state and holders won't be able to recover/transfer their tokens. At the same time, token owners could wait for ERC20 to be listed in Exchange, DEX, Uniswap or similar and then pause the token to avoid people withdrawing their tokens (I.E. when the price will go up).
Holders or investors must be careful and they must rely on centralized token governance.

Conclusion

The contracts don't contain very high or critical issues.
The code is inherited from OpenZeppelin contracts library so it should guarantee the compliance with the ERC20 standard.

Some improvements on custom code and logic could be made as suggested in the above audit.