SMART CONTRACT AUDIT REPORT

For

XGP Coin

July 2nd, 2021

Table of Content

• Disclaimer

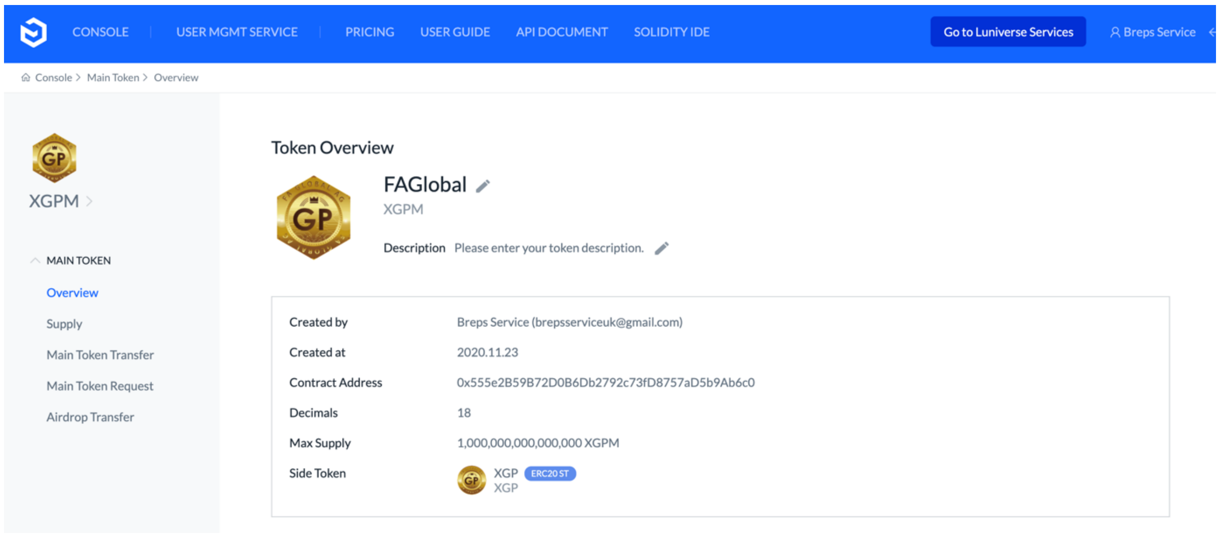
The audit makes no statements or warranties about the utility of the code, safety of the code, suitability of the business model, regulatory regime for the business model, or any other statements about the fitness of the contracts to purpose, or their bug-free status.

• Overview of Audit

This contract is deployed on Luniverse MainNet, and the contract type is coded in Ethereum ERC20 Solidity. XGP is a side token deployed on the Luniverse using Luniverse BaaS. This audit report only audits XGP written based on ERC20, excluding the Luniverse MainNet.

The XGP project to audit has 6 files. It contains approximately 500 lines of Solidity code. All the functions and state variables are well commented on using the natspec documentation, but that does not create any vulnerability.

• Overview of Token



Token Overview

FAGlobal
XGPM

Description Please enter your token description.

Created by	Breps Service (brepsserviceuk@gmail.com)
Created at	2020.11.23
Contract Address	0x555e2B59B72D0B6Db2792c73fD8757aD5b9Ab6c0
Decimals	18
Max Supply	1,000,000,000,000,000 XGPM
Side Token	XGP XGP

The following is information about XGP token obtained from [Luniverse Console.]

XGP

Token Type	ERC20
Created at	2020.11.23
Contract Address	0x555e2B59B72D0B6Db2792c73fD8757aD5b9Ab6c0
Decimals	18
Circulating Supply	50,000,000,000 XGP
Conversion Rate	1:1 (XGPM:XGP)

1. Attacks made to the contract

To check for the security of the contract, we tested several attacks to make sure that the contract is secure and follows best practices.

• Over and under flows

An overflow happens when the limit of the type of variable uint256, 2 ** 256, is exceeded. What happens is that the value resets to zero instead of incrementing more. On the other hand, an underflow happens when you try to subtract 0 minus a number bigger than 0. For example, if you subtract 0 - 1 the result will be = 2 ** 256 instead of -1. This is quite dangerous.

This contract **does** check for overflows and underflows by using OpenZeppelin's SafeMath to mitigate this attack, but all the functions have strong validations, which prevented this attack.

• Short address attack

If the token contract has enough tokens and the buy function doesn't check the length of the address of the sender, Ethereum's virtual machine will just add zeros to the transaction until the address is complete.

Although this contract **is not vulnerable** to this attack, there is some point where users can mess themselves due to this (Please see below). It is highly recommended to call functions after checking the validity of the address.

• Visibility & Delegatecall

It is also known as, The Parity Hack, which occurs while misusing of Delegatecall.

No such issues found in this smart contract and visibility also properly addressed. There are some places where there is no visibility defined. Smart Contract will assume "Public" visibility if there is no visibility defined. It is good practice to explicitly define the visibility, but again, the contract is not prone to any vulnerability due to this in this case.

• Reentrancy / TheDAO hack

Reentrancy occurs in this case: any interaction from a contract (A) with another contract (B) and any transfer of Ether hands over control to that contract (B). This makes it possible for B to call back into A before this interaction is completed.

Use of the "require" function in this smart contract mitigated this vulnerability.

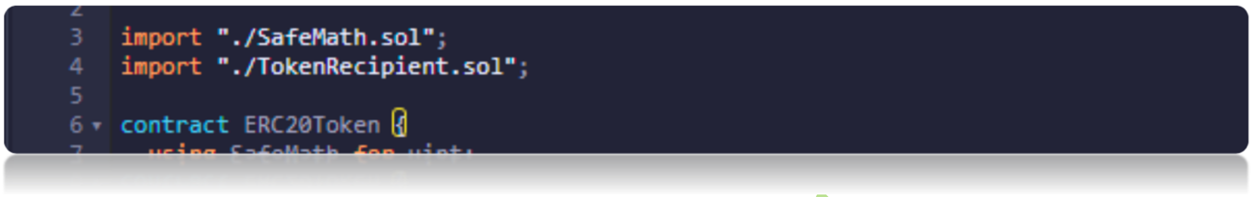
• Forcing Luniverse to a contract

While implementing "selfdestruct" in a smart contract, it sends all the Luniverse to the target address. Now, if the target address is a contract address, then the fallback function of the target contract does not get called. And thus, hackers can bypass the "Required" conditions. Here, the Smart Contract's balance has never been used as a guard, which mitigated this vulnerability.

2. Good things in Smart Contract

✓ SafeMath library:

- You are using SafeMath library it is a good thing. This protects you from underflow and overflow attacks.

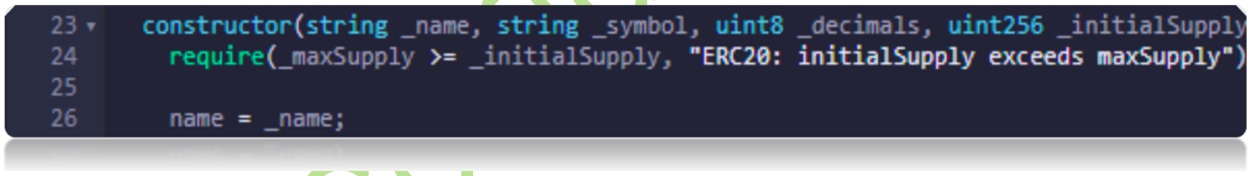


```
import "./SafeMath.sol";
import "./TokenRecipient.sol";

contract ERC20Token {
  using SafeMath for uint;
```

✓ Good required condition in functions:

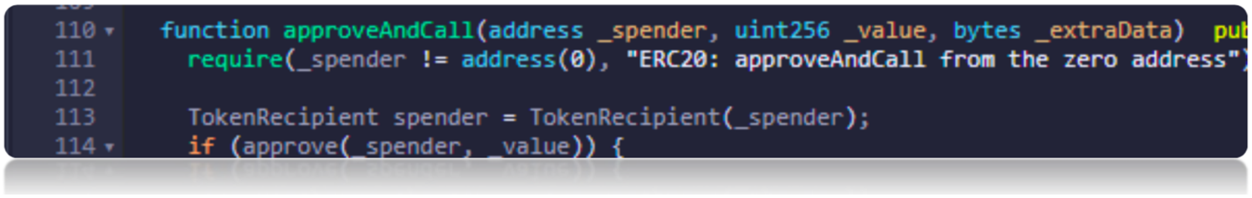
ERC20Token.sol

- Here you are checking that _initialSupply will not exceed _maxSupply value.



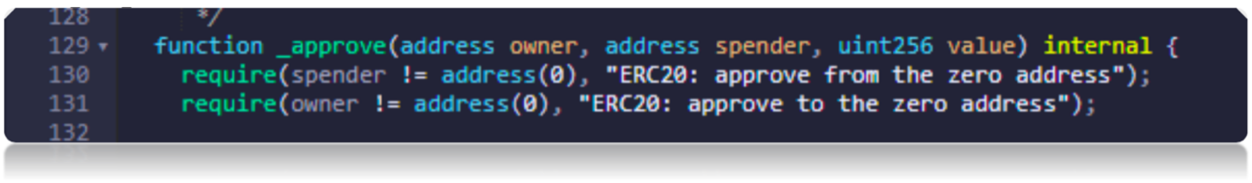
```
constructor(string _name, string _symbol, uint8 _decimals, uint256 _initialSupply
  require(_maxSupply >= _initialSupply, "ERC20: initialSupply exceeds maxSupply")

  name = _name;
```

- Here you are checking that _spender address value should be valid and proper value.



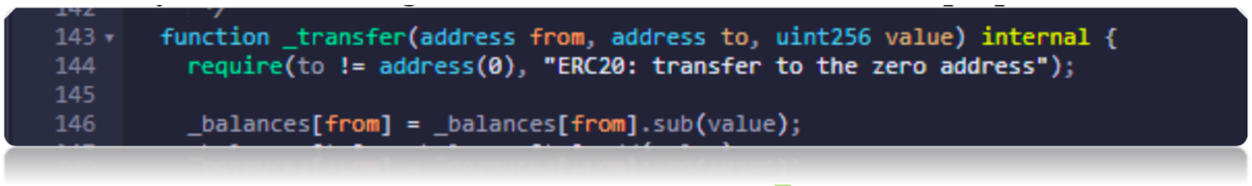
```
function approveAndCall(address _spender, uint256 _value, bytes _extraData)  pub
  require(_spender != address(0), "ERC20: approveAndCall from the zero address")

  TokenRecipient spender = TokenRecipient(_spender);
  if (approve(_spender, _value)) {
```

- Here you are checking that spender and owner addresses value are valid and proper.



```
function _approve(address owner, address spender, uint256 value) internal {
  require(spender != address(0), "ERC20: approve from the zero address");
  require(owner != address(0), "ERC20: approve to the zero address");
```

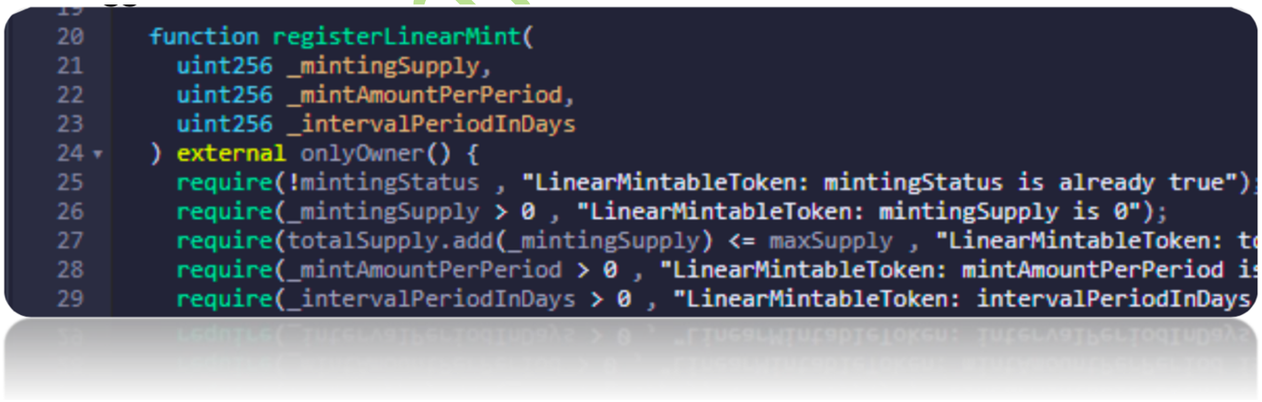
- Here you are checking that to address value is valid and proper.



```
function _transfer(address from, address to, uint256 value) internal {
  require(to != address(0), "ERC20: transfer to the zero address");

  _balances[from] = _balances[from].sub(value);
```

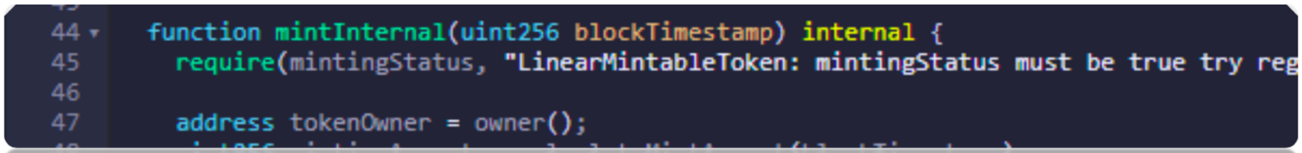
LinearMintableToken.sol

- Here you are checking that mintingStatus is not true,_mintingSupply should be bigger than 0, totalSupply + _mintingSupply should be smaller or equal to max_supply, _mintAmountPerPeriod and _intervalPeriodInDays should be bigger than 0.



```
function registerLinearMint(
  uint256 _mintingSupply,
  uint256 _mintAmountPerPeriod,
  uint256 _intervalPeriodInDays
) external onlyOwner() {
  require(!mintingStatus , "LinearMintableToken: mintingStatus is already true");
  require(_mintingSupply > 0 , "LinearMintableToken: mintingSupply is 0");
  require(totalSupply.add(_mintingSupply) <= maxSupply , "LinearMintableToken: t
  require(_mintAmountPerPeriod > 0 , "LinearMintableToken: mintAmountPerPeriod i
  require(_intervalPeriodInDays > 0 , "LinearMintableToken: intervalPeriodInDays
```

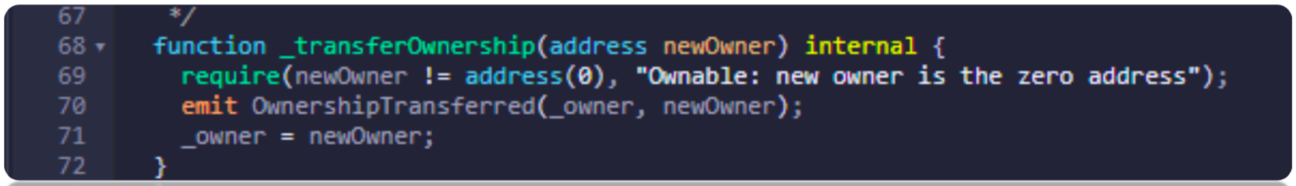
- Here you are checking that mintingStatus should be true.



```
function mintInternal(uint256 blockTimestamp) internal {
  require(mintingStatus, "LinearMintableToken: mintingStatus must be true try reg

  address tokenOwner = owner();
```

Ownable.sol

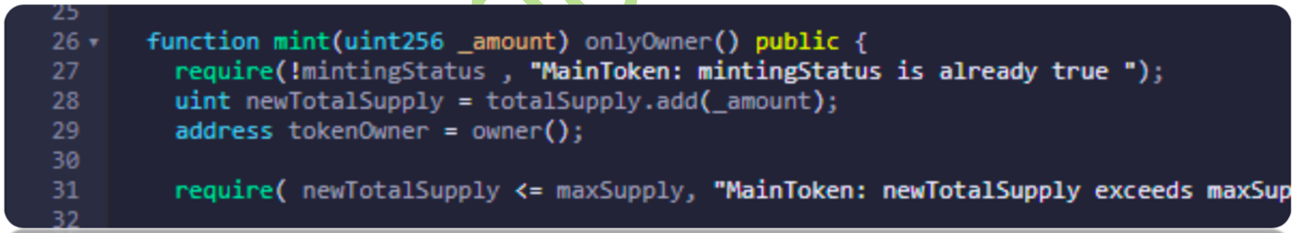
- Here you are checking that new Owner address value is valid and proper.



```
  */
function _transferOwnership(address newOwner) internal {
  require(newOwner != address(0), "Ownable: new owner is the zero address");
  emit OwnershipTransferred(_owner, newOwner);
  _owner = newOwner;
}
```

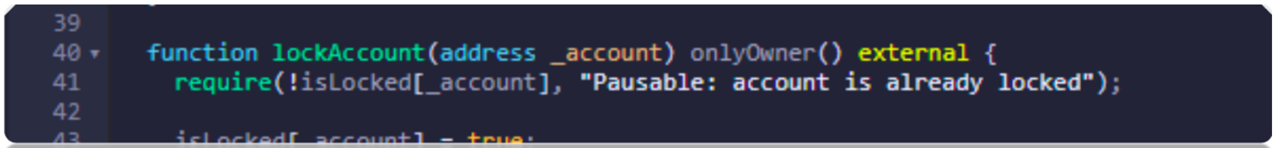
MainToken.sol

- Here you are checking that mintingStatus is already true or not.



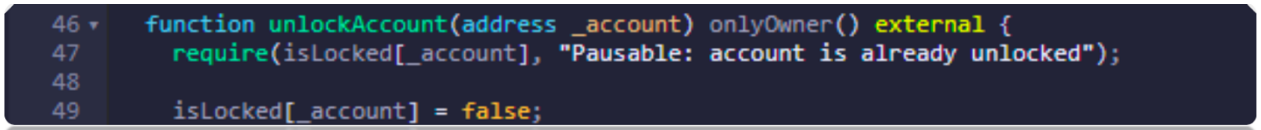
```
function mint(uint256 _amount) onlyOwner() public {
  require(!mintingStatus , "MainToken: mintingStatus is already true ");
  uint newTotalSupply = totalSupply.add(_amount);
  address tokenOwner = owner();

  require( newTotalSupply <= maxSupply, "MainToken: newTotalSupply exceeds maxSup
```

- Here you are checking that _accountaddress is not already locked.



```
function lockAccount(address _account) onlyOwner() external {
  require(!isLocked[_account], "Pausable: account is already locked");

  isLocked[_account] = true;
```

- Here you are checking that _accountaddress is not already unlocked.



```
function unlockAccount(address _account) onlyOwner() external {
  require(isLocked[_account], "Pausable: account is already unlocked");

  isLocked[_account] = false;
```

3. Severity Levels and explanation

Risk Level	Description
Critical	Critical vulnerabilities are severe as they are easy to exploit and can result in loss of tokens etc.
High	High level vulnerabilities are hard to exploit but can have an impact on smart contract execution, e.g., open access to an important function
Medium	Medium level vulnerabilities are important to fix but can't lead to loss of tokens
Low	Low-level vulnerabilities are mostly outdated code, unused code related. They have very low to no impact on smart contract execution

4. Critical vulnerabilities found in the contract

No Critical vulnerabilities found

5. Medium vulnerabilities found in the contract

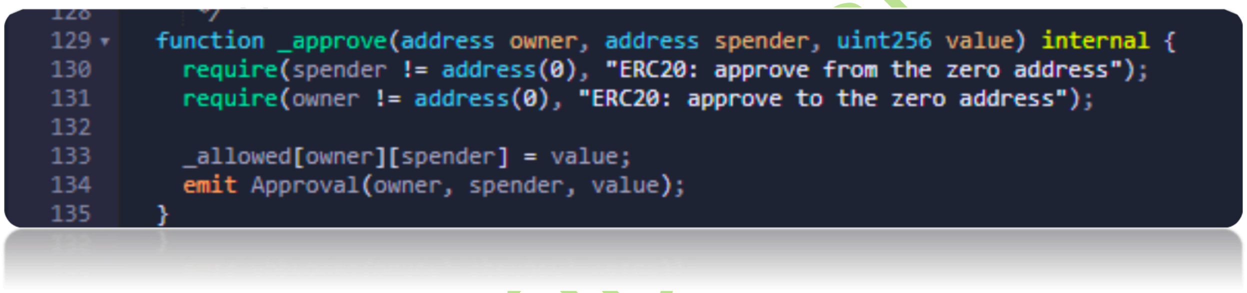
No Medium vulnerabilities found

6. Low severity vulnerabilities found

6.1 Check user balance in _approve in ERC20Token.sole :

We have found that _approve function you can are not checking user balance.

- In _approve function It is necessary to check that user can give allowance less or equal to their amount.
- There is no validation about user balance. So, it is good to check that a user did not set approval wrongly.
- **Function: - _approve**



```
function _approve(address owner, address spender, uint256 value) internal {
  require(spender != address(0), "ERC20: approve from the zero address");
  require(owner != address(0), "ERC20: approve to the zero address");

  _allowed[owner][spender] = value;
  emit Approval(owner, spender, value);
}
```

- Here you can check that balance of the sender should be bigger or equal to the amount value.

6.2 Compiler version is not fixed

- In this file, you have put "pragma solidity ^0.4.24;" which is not a good way to define the compiler version.
- Solidity source files indicate the versions of the compiler they can be compiled with. Pragma solidity >=0.4.24; // bad: compiles 0.4.24 and above pragma solidity 0.4.24; //good: compiles 0.4.24 only
- If you put(>=) symbol then you can get compiler version 0.4.24 and above. But if you don't use(^/>=) symbol then you are able to use only the 0.4.24 version. And if there are some changes that come in the compiler and you use the old version then some issues may come at deploy time.
- Try to use the latest version of solidity

7.Summary of the Audit

Overall, the code is well and performs well. There is no back door to steal funds.

Please try to check the address and value of the token externally before sending it to the solidity code.

Our final recommendation would be to pay more attention to the visibility of the functions, hardcoded address, and mapping since it's quite important to define who's supposed to execute the functions and to follow best practices regarding the use of assert, require, etc.

- **Good Point**: Code is written insecure way, all address and value validation done properly. Code performance and quality are good.
- **Suggestions:** Please try to use the static and latest version of solidity, check user balance in the _approve function.

**** This smart contract and audit report are not related to the US SEC(Securities and Exchange Commission).**

This document is an audit report that was analyzed the Smart Contract of XGP ERC20 Token on Luniverse and written by the X4Chain Auditing Team.

CEO of X4Chain, INC.

Charles Lee [illegible]