INTRODUCTION

This Audit Report focuses on the overall security, performance, deliverability of the NPTR Smart Contract, whose repository is available here

https://bitbucket.org/sricworkspace/sric.web3/src/main/contracts/

With this report, we have tried to ensure the reliability and correctness of the NPTR smart contract by a rigorous assessment of its architecture and the smart contract codebase.

— NPTR Audit Summary
— NPTR Audit
- PC Audit
Document information
Audit results
Audited target file
Vulnerability analysis
Vulnerability distribution
Summary of audit results
Contract file
Analysis of audit results

1. Unit tests passing, checking tests configuration (matching the configuration of the main network);
2. Compiler warnings;
3. Race Conditions. Reentrancy. Cross-function Race Conditions. Pitfalls in Race Condition solutions;
4. Possible delays in data delivery;
5. Transaction-Ordering Dependence (front running);
6. Timestamp Dependence;
7. Integer Overflow and Underflow;
8. DoS with (unexpected) Revert;
9. DoS with Block Gas Limit;
10. Call Depth Attack.
11. Methods execution permissions;
12. Oracles calls;
13. Economy model. It's important to forecast scenarios when a user is provided with additional economic motivation or faced with limitations. If application logic is based on an incorrect economy model, the application would not function correctly and participants would incur financial losses. This type of issue is most often found in bonus rewards systems.
14. The impact of the exchange rate on the logic;
15. Private user data leaks.

Project name : Bitxmi Token Contract
Project address: None
Code URL : https://bitbucket.org/sricworkspace/sric.web3/src/main/contracts/
Commit : None
Project target : NPTR Contracts Audit

Blockchain : Polygon Testnet
Test result : PASSED

EXECUTIVE SUMMARY

According to the assessment, the NPTR solidity smart contracts are ``well secured.

This audit is based of the the latest commit on the BitBucket repository mentioned above and viewable on Polygonscan with

https://mumbai.polygonscan.com/address/0x1Ea8B15af4Aa916C6Ba166Ca6dc6004771817E44

https://mumbai.polygonscan.com/address/0x98Dd7fe7503b75BB4E543886F81182171dc008ac

AUDIT GOALS

The focus of the audit was to verify that the Smart Contract System is secure, resilient and working according to the business logic specifications. The audit activities can be grouped in the following three categories:

Security
Identifying security related issues within each contract and the system of contract.

Sound Architecture
Evaluation of the architecture of this system through the lens of established smart contract best practices and general software best practices.

Code Correctness and Quality
A full review of the contract source code. The primary areas of focus include:

- Accuracy
- Readability
- Sections of code with high complexity

Weaknesses

Solidity	Call to the Unknown
	Gasless send
	Exception disorders
	Type casts
	Reentrancy
	Keeping secrets
EVM	Immutable bugs
	Tokens lost in transfer
	Stack size limit
Blockchain	Unpredictable state

Generating randomness
Time constraints

Security Considerations

Limit of Stack: here are some things to know about stack limit

- The maximum call stack depth is 1024
- External function calls fail if they exceed the maximum call stack of 1024 and throw an exception
- The low-level functions send (), call(), callcode (), and delegatecall () do not throw an exception but rather return false!

Keeping secrets: declaring a field as private does not guarantee does not guarantee its confidentiality. Since the blockchain is public, anyone can inspect the content of transactions and deduce the value of fields.

Immutable bugs: once a contract is published on the blockchain, it cannot be modified! If a contract contains a bug, there is no direct way to fix it.

Tokens lost in transfer: if tokens are sent to an orphan address, they will be lost forever. There has to be a way to verify the address is always correct.

Our solution: mapping each address in the custodial system to a user in our user database so that each wallet address is linked to a real person and their identity is verified by the back-end before tokens are sent.

Bad constructor calls: this could cause real problems especially when the wrong user calls a significant /critical function which may result in loss of funds or data.

Our solution: we have prevented this by disallowing unauthorized calls to critical functions by non-owner wallets.

Reentrancy attacK: this occurs when a function makes an external call to another untrusted contract. Then the untrusted contract makes a recursive call back to the original contract through a specific function in an attempt to drain funds.

When the contract fails to update its state before sending funds, the attacker can continuously drain funds by calling the withdraw function repeatedly.

Our solution: we are not allowing any external calls to our contract.

Transaction ordering dependency: this happens when

- The order if execution of function calls cannot be predicted
- There is no prior knowledge of the state of a contract during the execution call.

Our solution: to make sure transactions are well ordered to avoid sending funds before verification of balance and so on.

Integer overflow: this is when an uint (unsigned integer) reaches its byte size. Then the next element added will return the first-variable element.

Our solution: use safemath or Open-Zeppelin libraries with a later version of solidity from 0.8 which clearly addresses this issue.

Good Security Patterns Used

Access Restrictions
Intention: to restrict access to contract features based on appropriate criteria.
Applicability: we use the access restrictions when

- We want contracts to be called under certain conditions.
- We want to apply similar restrictions to multiple features.
- We want to strengthen the security against unauthorized access.

We can do this by using a modifier like require, onlyby, if or whichever way we choose.

Check of feets
Intention: this is to reduce the attack surface of malicious contracts that attempt to hijack control flow after an external call.
Applicability: we use this model to

- Avoid handling over control flow to an external entity.
- Protect against reentrancy attacks.

Emergency Stop
Intention: add an option to disable the emergency critical contract feature.
Applicability: we are suing this model to

- To suspend a contract in case of a hack
- To prevent critical functionality from abuse by undiscovered bugs
- To prepare for any sort failure from the contracts

THE AUDITING APPROACH AND METHODOLOGIES APPLIED

The following contracts are in scope for this audit:

1. SPR.sol
2. playerCounter.sol
3. houseWallet.sol

External dependencies and libraries (i.e OpenZeppelin standard libraries and any other contract are not in scope for this audit since they are industry standards.)

I performed rigorous testing of the project, starting with analyzing the code design patterns in which I reviewed the smart contract architecture to ensure it is structured and safe use of third-party smart contracts and libraries.

I then performed a formal line-by-line inspection of the Smart Contract to find any potential issues like race conditions, transaction-ordering dependence, timestamp dependence, and denial of service attacks.

The tests also include:

• Testing the functionality of the Smart Contract to determine proper logic has been followed throughout the whole process.

• Analyzing the complexity of the code in-depth and detailed, manual review of the code, line-by-line.

• Deploying the code on testnet using multiple clients to run live tests.

• Analyzing failure preparations to check how the Smart Contract performs in case of any bugs and vulnerabilities.

• Checking whether all the libraries used in the code are on the latest version.

• Analyzing the security of the on-chain data.

Document Information

Name	Auditor	Version	Date
NTPR Contracts	Amadasun Goodness	1.0.0	12-09-2022
NPTR Contracts	Amadasun Goodness	2.0.0	14-10-2022

Vulnerability Analysis

Vulnerability level	Number
Critical severity	0
High severity	0
Medium severity	0
Low severity	0

Summary of Audit Results

Vulnerability	Status
Unit tests	Safe
Compiler warnings	Safe
Race Conditions. Reentrancy	Safe
Possible delays in data delivery	Safe
Transaction-Ordering Dependence	Safe
Timestamp Dependence	Safe
Integer Overflow and Underflow	Safe
DoS with (unexpected) Revert;	Safe
DoS with Block Gas Limit;	Safe
Call Depth Attack	Safe
Methods execution permissions;	Safe
Oracles calls;	Safe
Economy model	Safe
The impact of the exchange rate on the logic	Safe
Private user data leaks.	Safe

SPR.sol

```
// SPDX-License-Identifier: MIT
pragma solidity >=0.4.22 <0.8.19;

//import openzepplin
import "./ERC20Upgradable.sol";

contract vest{
    function setDelegates(address, uint256, uint256) external returns(bool){}
    function lockToken(uint256, uint256, address) external returns (uint256){}
    function withdraw(uint256,address) external returns(bool){}
}
//begining of token contract
contract SPR is ERC20Upgradeable{
    address owner;
    mapping(bytes4 => bool) isRun;
    address public seedWallet;
    address public angelWallet;
    address public foundersWallet;
    address public airdropWallet;
    address public marketWallet;
    address public liquidityWallet;
    address public treasuryWallet;
    address public operationalWallet;
    address public dummyWallet;

    address public timeLock;

    uint256 _wei;
    uint256 vestId;
    uint256 public reserveMintingEndTime;
    vest _vest;



    function initialize(address _owner) public initializer {
        __ERC20_init("Sportiqo-Test5", "SPQ-Test5");
        owner = _owner; // set initial value in initialiowner = msg.sender;
        _wei= 1000000000000000000;
        reserveMintingEndTime = block.timestamp + (6 * (365 days));
    }
    function start() public returns(bool){
        isOwner();once();
        //ensure all the wallets has beens set
```

```
    require(seedWallet != address(0), "Seed wallet not set");
    require(angelWallet != address(0), "Angel wallet not set");
    require(foundersWallet != address(0), "Founders wallet not set");
    require(airdropWallet != address(0), "Airdrop wallet not set");
    require(marketWallet != address(0), "Market wallet not set");
    require(liquidityWallet != address(0), "Liquidity wallet not set");
    require(treasuryWallet != address(0), "Treasury wallet not set");
    require(operationalWallet != address(0), "Operational wallet not set");
    require(dummyWallet != address(0), "Dummy wallet not set");
    //mint 1B, the NPTR should have trannsfer ownership before this function call
    require(mint(1000000000)==true,"Minting is not done");
    //transfer 4% to seedWallet
    sendTo(owner,seedWallet, ((4 *1000000000) / 100) * _wei);
    // transfer 10% to angelWallet
    sendTo(owner,angelWallet, ((10 *1000000000) / 100) * _wei);
    //transfer 15 % to founderWallet
    sendTo(owner,foundersWallet, ((15 * 1000000000) / 100 ) * _wei) ;
    //transfer 2.997 % to airdropWallet
    sendTo(owner,airdropWallet, ((2.997 * 1000000000) / 100) * _wei);
    //transfer 6 % to marketWallet
    sendTo(owner,marketWallet, ((6 * 1000000000) / 100)* _wei);
    //transfer 12 % to liquidityWallet
    sendTo(owner,liquidityWallet, ((12 *1000000000) / 100)* _wei);
    //transfer 30 % to treasuryWallet
    sendTo(owner,treasuryWallet, ((30 *1000000000) / 100)* _wei);
    //transfer 20 % to operationalWallet
    sendTo(owner,operationalWallet, ((20 *1000000000) / 100)* _wei);
    //transfer 0.003% to dummy wallet
    sendTo(owner, dummyWallet, ((0.003 * 1000000000)/100)* _wei);

    return true;
}
//to start minting
function mint(uint256 amount) public  returns (bool){
    //to mint, only done by owner
    isOwner();isEnable();
    _mint(owner, amount * _wei);
    return true;
}
//to send tokens to
function sendTo(address from, address to, uint256 amount) public returns(bool){
    //to transfer minted tokens to
    isOwner();isEnable();
    require(balanceOf(from) >= amount || balanceOf(treasuryWallet) >= amount, "Insufficient amount");
    //remove tokens from smart contract wallet
```

```
    // if(balanceOf(from) >= amount){
    //     _balances[from] = _balances[from] - amount;
    // }else if(balanceOf(treasuryWallet) >= amount){
    //     _balances[treasuryWallet] = _balances[treasuryWallet] - amount;
    // }
    // //send tokens to wallet
    // _balances[to] = _balances[to] + amount;

    _transfer(from, to, amount);
    return true;
}
//to burn tokens only by  ADMIN
function burn(uint256 amount) public  returns (bool){
    isOwner();isEnable();
    require(_balances[address(this)] >= amount, "Insufficient amount");
    _burn(address(this), amount);
    return true;
}

//to disable and enable token
function enableToken(bool _value) public returns (bool){
    isOwner();
    _enable(_value);
}
//to transfer ownership
function makeAdmin(address _address) public returns (bool){
    //transfer ownership status
    isOwner();
    owner = _address;
    return true;
}

//get available minted tokens
function mintedTokens() public view returns (uint256 bal){
    //transfer ownership status
    bal = balanceOf(address(this));
    return bal;
}
//to change the tradding wallet or reward wallet
function setAddress(string memory typex, address _address) public returns(bool){
    isOwner();
    if(strequ(typex,"seed")){
        seedWallet = _address;
    }else if(strequ(typex, "angel")){
        angelWallet = _address;
```

```
        }else if(strequ(typex, "founders")){
            foundersWallet = _address;
        }else if(strequ(typex, "airdrop")){
            airdropWallet = _address;
        }else if(strequ(typex, "market")){
            marketWallet = _address;
        }else if(strequ(typex, "liquidity")){
            liquidityWallet = _address;
        }else if(strequ(typex, "treasury")){
            treasuryWallet = _address;
        }else if(strequ(typex, "operations")){
            operationalWallet = _address;
        }else if(strequ(typex, "dummy")){
            dummyWallet = _address;
        }
        return true;
    }

    //auxillary functions
    function strequ(string memory a, string memory b) private pure returns (bool) {
        if(bytes(a).length != bytes(b).length) {
            return false;
        } else {
            return keccak256(abi.encodePacked(a)) == keccak256(abi.encodePacked(b));
        }
    }
    //check ownership
    function isOwner() public returns(bool){
        require(owner == msg.sender, "Can only be done by Owner");
        return true;
    }
    //this modifier allows this function to be done once
    function once() private returns (bool){
        require(isRun[msg.sig] == false, "This method has already being called");
        //set it to as being called
        isRun[msg.sig] = true;
        return true;
    }
}
```

playerCiunter.sol

```
// SPDX-License-Identifier: MIT
pragma solidity >=0.4.22 <0.8.19;
import "./ERC20.sol";
import "./Pausable.sol";

//begining of token contract
contract playerCounter is ERC20, Pausable{
    address owner;
    address houseAddr;
    uint256 _wei = 1000000000000000000;

    constructor(string memory _name, string memory _symbol, address _houseAddr, uint256 _totalSupply)
ERC20(_name, _symbol) {

        owner = _houseAddr; // set initial value in initializer

        houseAddr = _houseAddr; //set house wallet address

        _mint(houseAddr, _totalSupply*_wei);
    }
    //to start minting, to be done by PC ADMIN
    function mint(uint256 amount) public  returns (bool){
        //to mint, only done by owner
        isOwner();
        _mint(houseAddr, amount*_wei);
        return true;
    }
    //to send tokens to
    function sendTo(address from, address to, uint256 amount) public returns(bool){
        //to transfer minted tokens to
        isOwner();



        _transfer(from, to, amount);
        return true;
    }



    function deactivate() public {
        isOwner();
        _pause();
    }
```

```
    function activate() public  {
        isOwner();
        _unpause();
    }

    //to burn tokens only by PC ADMIN
    function burn(address from) public  returns (bool){
        isOwner();
        require(_balances[from] >0, "Insufficient amount");
        _burn(from, _balances[from]);
        return true;
    }


    //to transfer ownership
    function makeAdmin(address _address) public returns (bool){
        //transfer ownership status
        isOwner();
        owner = _address;
        return true;
    }

    //get available minted tokens
    function mintedTokens() public view returns (uint256 bal){
        //transfer ownership status
        bal = balanceOf(houseAddr);
        return bal;
    }

    //auxillary functions
    function isOwner() internal virtual returns(bool){
        require(owner == msg.sender, "Can only be done by Owner");
        return true;
    }

    function _beforeTokenTransfer(address from, address to, uint256 amount)
    internal
    whenNotPaused
    override
    {
        super._beforeTokenTransfer(from, to, amount);
    }
}
```

houseWallet.sol

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.4;

//this contract acts an an escrow for PCs
contract ERC20 {
   //all ERC20 tokens has balance of
   function balanceOf(address) external view  returns (uint256){}
   function transfer(address,uint256) external returns (bool) {}
   function escrowClaim(address, address, uint256) external returns (bool) {}
}
contract HouseWallet{

   ERC20 token;
   address owner;

   //events declaration
   event sent(address to, uint256 amount);
   event received(address from, uint256 amount);

   constructor(){
      owner = msg.sender;
   }
   //to send PCs tokens to a recipient
   function send(address to, uint256 amount, address _pc) public returns(bool){
      isOwner();
      token = ERC20(_pc);
      //check if it has enough amount to do transfer
      require(token.balanceOf(address(this)) >= amount, "Not sufficient PCs");
      //transfer token to user
      token.transfer(to, amount);
      emit sent(to, amount);
      return true;
   }

   //to receive PCs tokens from users
   function claim(address from, uint256 amount, address _pc) public returns(bool){
      token = ERC20(_pc);
      //check if user has enough amount to do transfer
      require(token.balanceOf(msg.sender) >= amount, "Not sufficient PCs");
      //transfer token to user
      token.escrowClaim(from, address(this), amount);
      emit received(from, amount);
      return true;
```

```
    }

    //to transfer ownership
    function makeOwner(address _address) public returns (bool){
        //transfer ownership status
        isOwner();
        owner = _address;
        return true;
    }
    //check ownership
    function isOwner() public returns(bool){
        require(owner == msg.sender, "Can only be done by Owner");
        return true;
    }

}
```

ANALYSIS OF AUDIT RESULTS

The audit reports are as described below:

1. Unit Tests: the unit tests were conducted on the SPR.sol and playerCounter.sol smart contract files. Testing was carried out using the hardhat and ethers packages in a NodeJs environment, as well as the web3j library in a Java environment which generates an ABI build of the smart contract, that is then used for initiating RPC calls to different functions on the smart contract for checking whether the functions execute healthily as expected.

The different tests were:

a) SafeMath Library Test: the SafeMath library is responsible for carrying out arithmetic operations needed by the smart contract. Tests carried out on the SafeMath library returned a successful response on all the five arithmetic functions of add, sub, mul, div, and mod.

b) Ownable, ERC20 Contracts and the IERC20 interface were also tested, all of which returned a successful response.

c) SPR and playerCounter contract: these contracts provide the main methods that execute instructions inherited from the SafeMath library, as well as the Ownable, Pausable, and ERC20 contracts and the IERC20 interface. The constructor of the SPR was tested here to check that the initial total token supply matched the expected token supply, as well as the start(), mint(), sendTo(), burn(), enableToken(), mintedTokens(), setAddress(), strequ() functions. All of the PlayerCounter.sol's mint(), sendPcToUser(), transferPc(), and decreaseSupply() functions were tested alongside the sol contract. The responses returned were entirely positive

.

Result: PASSED!

2. Compiler Warnings: there were no compilation warnings encountered. This is a direct consequence of the unit testing actions taken above. Otherwise unit testing wouldn't be possible.

Also, the Remix online editor was used to compile the contracts before deploying them on Matic Testnet and no errors or warnings were encountered.

Result: PASSED!

3. Race Condition. Reentrancy. Cross-function Race Conditions. Pitfalls in Race Conditions: upon rigorous evaluation of the smart contracts, no problem of Race Condition, Reentrancy, or Cross-function Race Conditions was discovered

Result: PASSED!

4. Possible Delays in Data Delivery: There may be possible delays in the delivery of data in a smart contract and can cause different issues to come up. AFter tests, no issue(s) found here.

Result: PASSED!

5. Transaction-Ordering Dependence: The combination of external calls to other contracts and the multi-user nature of the underlying blockchain gives rise to a variety of potential Solidity pitfalls whereby users race code execution to obtain unexpected states. Re- Entrancy is one example of such a race condition. In this section we will talk more generally about different kinds of race conditions that can occur on the blockchain. There are a variety of good posts on this subject, a few are: Wiki - Safety, DASP - Front-Running and the Consensus - Smart Contract Best Practices. After tests, no issue(s) found here.

Result: PASSED!

6. Timestamp Dependence: Block timestamps have historically been used for a variety of applications, such as entropy for random numbers (see the Entropy Illusion section for further details), locking funds for periods of time and various state-changing conditional statements that are time-dependent. Miner's have the ability to adjust timestamps slightly which can prove to be quite dangerous if block timestamps are used incorrectly in smart contracts. After tests, no issues(s) found here.

Result: PASSED!

7. Integer Overflow and Underflow: The Ethereum Virtual Machine (EVM) specifies fixed-size data types for integers. This means that an integer variable only has a certain range of numbers it can represent. A uint8 for example, can only store numbers in the range [0,255]. Trying to store 256 into a uint8 will result in 0. If care is not taken, variables in Solidity can be exploited if user input is unchecked and calculations are performed which result in numbers that lie outside the range of the data type that stores them. After test, no issues(s) found here

Result: PASSED!

8. DoS with (unexpected) Revert: This category is very broad, but fundamentally consists of attacks where users can leave the contract inoperable for a small period of time, or in some cases, permanently. This can trap Blockchain Currency in these contracts forever, as was the case with the Second Parity MultiSig hack. After tests, no issues(s) found here.

Result: PASSED!

9. DoS with Block Gas Limit: This has to do with attacks when users can't access functions due to gas limit. After test, no issues(s) found here

Result: PASSED!

10. Call Depth Attack: there are a number of ways of performing external calls in solidity. Sending Blockchain Currency to external accounts is commonly performed via the transfer() method. However, the send() function can also be used and, for more versatile external calls, the CALL opcode can be directly employed in solidity. The call() and send() functions return a boolean indicating if the call succeeded or failed. Thus these functions have a simple caveat, in that the transaction that executes these functions will not revert if the external call (initialized by call() or send()) fails, rather the call() or send() will simply return false. A common pitfall arises when the return value is not checked, rather the developer expects a revert to occur. After tests, no issue(s) found here.

Result: PASSED!

11. Methods execution permissions: Contract managers who can control liquidity or pledge pools, etc., or impose unreasonable restrictions on other users. After tests, no issue(s) found here.

Result: PASSED!

12. Oracle calls: The CALL and DELEGATECALL opcodes are useful in allowing developers to modularise their code. Standard external message calls to contracts are handled by the CALL opcode whereby code is run in the context of the external contract/function. The DELEGATECALL opcode is identical to the standard message call, except that the code executed at the targeted address is run in the context of the calling contract along with the fact that msg.sender and msg.value remain unchanged. This feature enables the implementation of libraries whereby developers can create reusable code for future contracts. After tests, no issue(s) found here

Result: PASSED!

13. Economy model: The economy model has been checked from the tokenomics to tokens and fee distribution. After tests, no issue(s) found here.

Result: PASSED!

14. Impact of exchange rate on smart contract login: Typically when Blockchain Currency is sent to a contract, it must execute either the fallback function, or another function described in the contract. There are two exceptions to this, where Blockchain Currency can exist in a contract without having executed any code. Contracts which rely on code execution for every Blockchain Currency sent to the contract can be vulnerable to attacks where Blockchain Currency is forcibly sent to a contract. After tests, no issue(s) found here.

Result: PASSED!

15. Private user data leaks: This is to check if the data of the user has possible leak vulnerabilities in the smart contract. After tests, no issue(s) found here

Result: PASSED!

DISCLAIMER

This report is not an endorsement or indictment of any particular project or team, and the report does not guarantee the security of any particular project. This report does not consider, and should not be interpreted as considering or having any bearing on, the potential economics of a token, token sale or any other product, service or other asset. Cryptographic tokens are emergent technologies and carry with them high levels of technical risk and uncertainty. This report does not provide any warranty or representation to any Third-Party in any respect, including regarding the bugfree nature of code, the business model or proprietors of any such business model, and the legal compliance of any such business. No third party should rely on this report in any way, including for the purpose of making any decisions to buy or sell any token, product, service or other asset. Specifically, for the avoidance of doubt, this report does not constitute investment advice, is not intended to be relied upon as investment advice, is not an endorsement of this project or team, and it is not a guarantee as to the absolute security of the project. I owe no duty to any Third-Party by virtue of publishing these Reports.

The scope of my review is limited to a review of Solidity code and only the Solidity code noted as being within the scope of the review within this report. The Solidity language itself remains under development and is subject to unknown risks and flaws. The review does not extend to the compiler layer, or any other areas beyond Solidity that could present security risks. Cryptographic tokens are emergent technologies and carry with them high levels of technical risk and uncertainty.

This audit does not give any warranties on finding all possible security issues of the given smart contracts, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit-based assessment cannot be considered comprehensive, I always recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contracts.