Smart Contracts Security Audit

Skybridge

2021-04-20

Content

1. Introduction

Swingby is a decentralized proof-of-stake network that uses great advancements in cryptography research to allow you to move your tokens onto other chains without a trusted party.

Swingby's Skybridge uses the latest in threshold signature cryptography ("TSS") and multi-party computing ("MPC") research to deliver a layer-2 cross-chain bridging protocol for users to move their assets from one blockchain to another. Each bridge exists as a proof-of-stake network.

As requested by SwingBy and as part of the vulnerability review and management process, Red4Sec has been asked to perform a security code audit and a cryptographic assessment in order to evaluate the security of the Skybridge Smart Contracts source code.

2. Disclaimer

This document only represents the results of the code audit conducted by Red4Sec Cybersecurity and should not be used in any way to make investment decisions or as investment advice on a project.

Likewise, the report should not be considered neither "endorsement" nor "disapproval" of the guarantee of the correct business model of the analyzed project.

3. Scope

The Skybridge review includes the solidity smart contracts from skybridge-contract GitHub repository, commit `87742e21df7fb62359104502f29f69d173379318`.

4. Conclusions

To this date, 20th of April 2021, the general conclusion resulting from the conducted audit is that **Skybridge's smart contracts are secure** and do not present any critical-high known vulnerabilities, although Red4Sec has found a few potential improvements.

A few low impact issues were detected where an action plan must be elaborated to guarantee its resolution to help SwingBy improve the security and quality of its developments. The quality of the unit test could be improved, some unit tests do not correctly check all the results to ensure they are working satisfactorily.

It must be mentioned that the Skybridge contracts have control over the transactions register, the swaps and the fees. Additionally, they contain functionality for the fee's distribution and the change of nodes. However, the verification of the authenticity and legitimacy of the stored transactions, their amounts and properties, is done outside of the contract by the SwingBy nodes, who control the owner's Threshold Signatures.

It is in the process of modifying the Owner of the **SwapContract** contract where some discrepancies have been found and the SwingBy team must study them to verify that they comply with the expected logic.

5. Issues & Recommendations

Owner Management

The functionalities of the **SwapContract** contract are mainly controlled by the TSS validators, which are essentially controlled by the Skybridge project, without these the contract would be fully inoperative. Therefore, the *renounceOwnership* method inherited from the **Ownable** class, should not be accessible, because in case it is invoked by the owners it would leave the contract inoperative.

The same problem occurs with the *transferOwnership* method. The contract already has a specific method for changing Owners, *churn()*, which also applies certain verifications, so the *transferOwnership* method should not be accessible and should not allow the change of Owner without performing the same verifications.

The Skybridge project delegates the control of the owners to a group of Threshold Signature Scheme validators, that rotate over time. If at any given moment there was a malicious subgroup of validators that reached the minimum Threshold, they could use the methods described above to take full control of the contract.

Source References

- https://github.com/SwingbyProtocol/skybridge-contract/blob/7f97b2aba97e704315c7b038d6a86d99964684c6/contracts/SwapContract.sol#L321
- https://github.com/OpenZeppelin/openzeppelin-contracts/blob/cec0800c541c809f883a37f2dfb91ec4c90263c5/contracts/access/Ownable.sol#L54
- https://github.com/OpenZeppelin/openzeppelin-contracts/blob/cec0800c541c809f883a37f2dfb91ec4c90263c5/contracts/access/Ownable.sol#L63

Reentrancy

The Reentrancy attack is an Ethereum vulnerability which occurs when external contract calls can make new calls to the calling contract before the initial execution is completed. For a function, this means that the contract state could change in the middle of its execution as a result from a call to an untrusted contract or the use of a low-level function with an external address.

The *distributeNodeRewards* function of the **SwapContract** contract sets the *lockedLPTokensForNode* and *freesLPTokensForNode* variables to *0* after the funds are sent by the contract. Theoretically, this could allow an attacker that is using a smart contract to call this function so that it can be called back on its fallback method; when the variables are established after the call, the next call will be accepted as valid, producing a reentrancy attack and allowing the attacker to drain the contract's funds.

Although the **lpToken** is initially reliable and controlled by the project, so that it acts according to the expected logic, it is always convenient to apply protections in case we use third-party tokens in the future.

This sort of mistake may be avoided in the future by always establishing the verification flags before a contract's call.

```
function distributeNodeRewards() external override returns (bool) {
    // Reduce Gas
    uint256 rewardLPTsForNodes =
        lockedLPTokensForNode.add(feesLPTokensForNode);
    require(
        rewardLPTsForNodes > 0,
        "totalRewardLPsForNode is not positive"
    );
    bytes32[] memory nodeList = getActiveNodes();
    uint256 totalStaked = 0;
    for (uint256 i = 0; i < nodeList.length; i++) {
        totalStaked = totalStaked.add(
            uint256(uint96(bytes12(nodeList[i])))
        );
    }
    IBurnableToken(lpToken).mint(address(this), lockedLPTokensForNode);
    for (uint256 i = 0; i < nodeList.length; i++) {
        IBurnableToken(lpToken).transfer(
            address(uint160(uint256(nodeList[i]))),
            rewardLPTsForNodes
                .mul(uint256(uint96(bytes12(nodeList[i]))))
                .div(totalStaked)
        );
    }
    emit DistributeNodeRewards(rewardLPTsForNodes);
    lockedLPTokensForNode = 0;
    feesLPTokensForNode = 0;
    return true;
}
```

Source references

- https://github.com/SwingbyProtocol/skybridge-contract/blob/7f97b2aba97e704315c7b038d6a86d99964684c6/contracts/SwapContract.sol#L286-L287

The same can happen with the *_issueLPTokensForFloat* method of the **SwapContract** contract, in this case, it is recommended to send the LP tokens once the corresponding amount has been added. Although there is a whitelist, this will allow to avoid malicious tokens created by a possible attacker with the aim of profiting financially.

```
// Send LP tokens to LP
IBurnableToken(lpToken).mint(to, amountOfLP.sub(depositFees));
// Add deposit fees
lockedLPTokensForNode = lockedLPTokensForNode.add(depositFees);
// Add float amount
_addFloat(_token, amountOfFloat);
_addUsedTx(_txid);
emit IssueLPTokensForFloat(
    to,
    amountOfFloat,
    amountOfLP,
    nowPrice,
    depositFees,
    _txid
);
return true;
```

Source references

- https://github.com/SwingbyProtocol/skybridge-contract/blob/7f97b2aba97e704315c7b038d6a86d99964684c6/contracts/SwapContract.sol#L468

Recommendations

- Assure all internal state changes are performed before external calls are executed. This is known as the Checks-Effects-Interactions pattern.
- Use a reentrancy lock (ie. OpenZeppelin's ReentrancyGuard).

Wrong Logic for _swap method

The *_swap* function of the **SwapContract** contract incorrectly performs the swap operation between the origin and the destination, a subtraction of the balance of the destination and a sum of the balance of the origin are performed, when in any swap operation they must be carried out the other way around.

Below, we can see how the use of the *_dest* and *_source* prefixes can be confusing and potentially lead to errors.

```
/// @dev _swap collects swap amount to change float.
/// @param _sourceToken The address of source token
/// @param _destToken The address of target token.
/// @param _swapAmount The amount of swap.
function _swap(
    address _sourceToken,
    address _destToken,
    uint256 _swapAmount
) internal {
    floatAmountOf[_destToken] = floatAmountOf[_destToken].sub(
        _swapAmount,
        "_swap: float amount insufficient"
    );
    floatAmountOf[_sourceToken] = floatAmountOf[_sourceToken].add(
        _swapAmount
    );
    emit Swap(_sourceToken, _destToken, _swapAmount);
}
```

Additionally, it has been possible to verify that this function only appears covered in the unit tests by the *"collectSwapFeesForBTC test"*, however in this test the expected values are not verified after the call to the *collectSwapFeesForBTC* method, so it would be convenient to correct said test to carry out an adequate coverage and check the correct operation of the call.

Source references

- https://github.com/SwingbyProtocol/skybridge-contract/blob/7f97b2aba97e704315c7b038d6a86d99964684c6/contracts/SwapContract.sol#L595-L601
- https://github.com/SwingbyProtocol/skybridge-contract/blob/7f97b2aba97e704315c7b038d6a86d99964684c6/test/testSwapContract.js#L100

Insecure Transfers

The ERC-20[1] standard specifies that the *transfer* and *transferFrom* functions with the result of this operation, will return a boolean.

The contract does not contemplate this result, although it is true that most of the token ERC-20 implementations make a revert if these methods fail, the result of external contract calls should always be checked.

It is essential to check in all the transfers that the returned value is true, it is also possible to use SafeERC20[2] from Open Zeppelin contracts, which already makes the check after the execution of transfers.

Source references

- https://github.com/SwingbyProtocol/skybridge-contract/blob/7f97b2aba97e704315c7b038d6a86d99964684c6/contracts/SwapContract.sol#L278

Incorrect Pragma

In the audited contracts, the pragma establishes that they are compatible with different versions of Solidity. However, it has been verified that the code is only compatible with the *0.7.x* branch of Solidity and contains instructions that are not backward compatible with the *0.6* branch nor with superior versions.

```
// SPDX-License-Identifier: AGPL-3.0
pragma solidity >=0.6.0 <0.8.0;
```

Therefore, it is advisable to modify the smart contract's pragma to the *0.7* version and to use the last version of the compiler for this branch, to the date of this report, the *0.7.6*, not the *0.7.5* as its currently established in *truffle-config.js*.

References

- https://github.com/ethereum/solidity/blob/develop/Changelog.md

[1] https://eips.ethereum.org/EIPS/eip-20

[2] https://github.com/OpenZeppelin/openzeppelin-contracts/blob/8b58fc71919efda463e53b3ffa083edac19c85b8/contracts/token/ERC20/SafeERC20.sol#L72

Gas Optimization

Software optimization is the process of modifying a software system to make an aspect of it work more efficiently or use less resources. This premise must be applied to smart contracts as well, so that they execute faster or in order to save GAS.

On Ethereum blockchain, GAS is an execution fee which is used to compensate miners for the computational resources required to power smart contracts. If the network usage is increasing, so will the value of GAS optimization.

These are some of the requirements that must be met to reduce GAS consumption:

- Short-circuiting.
- Remove redundant or dead code.
- Delete unnecessary libraries.
- Explicit function visibility.
- Use of proper data types.
- Use hard-coded CONSTANT instead of state variables.
- Avoid expensive operations in a loop.
- Pay special attention to mathematical operations and comparisons.

Storage Optimization

The use of the *inmutable*[3] keyword is recommended to obtain less expensive executions in the future, by having the same behaviour as a constant. However, by defining its value in the constructor we have a significant save of GAS.

This behaviour has been observed in:

- https://github.com/SwingbyProtocol/skybridge-contract/blob/7f97b2aba97e704315c7b038d6a86d99964684c6/contracts/SwapContract.sol#L21-L28
- https://github.com/SwingbyProtocol/skybridge-contract/blob/7f97b2aba97e704315c7b038d6a86d99964684c6/contracts/BurnableToken.sol#L14-L18

[3] https://docs.soliditylang.org/en/v0.6.5/contracts.html#immutable

Dead Code

In programming, a part of the source code that is executed but it is never used is known as dead code. The execution of this type of code consumes more GAS during deployment in something that is never used.

The *_burnFrom* function from the **BurnableToken** contract is never used and can be removed.

```
function _burnFrom(address account, uint256 amount) internal {
    _burn(account, amount);
    _approve(
        account,
        _msgSender(),
        allowances[account][_msgSender()].sub(
            amount,
            "ERC20: burn amount exceeds allowance"
        )
    );
}
```

Execution Cost

Following there are some possible code optimizations that will make the audited smart contracts more low-cost and consequently more optimal and accessible to users.

1. In the *churn* function of the **SwapContract** contract, the variables *_nodeRewardsRatio* and *_withdrawalFeeBPS* are of *unit8* type, therefore it is not necessary to check that their value is greater than or equal to zero since it can never be the opposite.

```
require(
    _nodeRewardsRatio >= 0 && _nodeRewardsRatio <= 100,
    "_nodeRewardsRatio is not valid"
);
require(
    _withdrawalFeeBPS >= 0 && _withdrawalFeeBPS <= 100,
    "_withdrawalFeeBPS is invalid"
);
```

Source references

- https://github.com/SwingbyProtocol/skybridge-contract/blob/7f97b2aba97e704315c7b038d6a86d99964684c6/contracts/SwapContract.sol#L338-L345

2. In the *getActiveNodes* method of the **SwapContract** contract it is possible to save GAS by avoiding the use of safemath, this function is always recommended to perform mathematical operations, but in this specific case it is used as a counter, and it is impossible to produce an overflow before an out of gas, so avoiding its use would save a significant gas cost.

```
function getActiveNodes() public view override returns (bytes32[] memory) {
    uint256 nodeCount = 0;
    uint256 count = 0;
    // Seek all nodes
    for (uint256 i = 0; i < nodeAddrs.length; i++) {
        if (nodes[nodeAddrs[i]] != 0x0) {
            nodeCount = nodeCount.add(1);
        }
    }
```

Source references

- https://github.com/SwingbyProtocol/skybridge-contract/blob/7f97b2aba97e704315c7b038d6a86d99964684c6/contracts/SwapContract.sol#L428

3. In the **SwapContract** contract it is declared in the *convertScale* variable. A GAS saving could be made by declaring it as constant, considering that all BTC wrapped tokens have 8 decimal and their value will not change, additionally, this value could be required to be this way during the creation of the contract.

```
// Set convertScale
convertScale = 10**(IERC20(_btct).decimals() - 8);
```

Source references

- https://github.com/SwingbyProtocol/skybridge-contract/blob/7f97b2aba97e704315c7b038d6a86d99964684c6/contracts/SwapContract.sol#L94

4. In the *getActiveNodes* method of the **SwapContract** contract it is possible to save a loop with a consequent GAS saving that it is generated by pushing the array.

```
/// @dev getActiveNodes returns active nodes list (stakes and amount)
function getActiveNodes() public view override returns (bytes32[] memory) {
    uint256 nodeCount = 0;
    uint256 count = 0;
    // Seek all nodes
    for (uint256 i = 0; i < nodeAddrs.length; i++) {
        if (nodes[nodeAddrs[i]] != 0x0) {
            nodeCount = nodeCount.add(1);
        }
    }
    bytes32[] memory _nodes = new bytes32[](nodeCount);
    for (uint256 i = 0; i < nodeAddrs.length; i++) {
        if (nodes[nodeAddrs[i]] != 0x0) {
            _nodes[count] = nodes[nodeAddrs[i]];
            count = count.add(1);
        }
    }
    return _nodes;
}
```

Source references

- https://github.com/SwingbyProtocol/skybridge-contract/blob/7f97b2aba97e704315c7b038d6a86d99964684c6/contracts/SwapContract.sol#L422-L439

5. In the *collectSwapFeesForBTC* function of the **SwapContract** contract, there is an argument that must always be *0x00..*, so it is advisable to remove this argument, and perform the checks in the caller of the function, since this argument is not used in the method.

```
function collectSwapFeesForBTC(
    address _destToken,
    uint256 _incomingAmount,
    uint256 _minerFee,
    uint256 _rewardsAmount
) external override onlyOwner returns (bool) {
    require(_destToken == address(0), "_destToken should be address(0)");
    address _feesToken = BTCT_ADDR;
```

Source References

- https://github.com/SwingbyProtocol/skybridge-contract/blob/7f97b2aba97e704315c7b038d6a86d99964684c6/contracts/BurnableToken.sol#L342

- https://github.com/SwingbyProtocol/skybridge-contract/blob/7f97b2aba97e704315c7b038d6a86d99964684c6/contracts/SwapContract.sol#L197

6. The *singleTransferERC20* and *_addNode* methods of the **SwapContract** contract return a boolean but a false is never returned, so it would be convenient to eliminate the return, since there will be no possibility of obtaining a value other than true.

   Source References
   - https://github.com/SwingbyProtocol/skybridge-contract/blob/7f97b2aba97e704315c7b038d6a86d99964684c6/contracts/SwapContract.sol#L144
   - https://github.com/SwingbyProtocol/skybridge-contract/blob/7f97b2aba97e704315c7b038d6a86d99964684c6/contracts/SwapContract.sol#L672

Code Style Improvement

During the development of any software or application, it is necessary to follow standards of coding best practices and code readability, helping the initial development and the subsequent maintenance and enhancement by people other than the original authors.

It has been found that a few segments of the code do not follow the best developing practices, this is not a vulnerability by itself, but by fixing it the code will improve and it will reduce the chances of new vulnerabilities emerging.

Use of statements without reason message
It was verified that the reason message is not specified in some revert/require instructions, in order to give the user more information, which consequently makes it more user friendly.

An example of this issue can be found in the following methods:

- *recordIncomingFloat*
- *_safeTransfer*
- *_issueLPTokensForFloat*
- *_addFloat*

Source References

- https://github.com/SwingbyProtocol/skybridge-contract/blob/7f97b2aba97e704315c7b038d6a86d99964684c6/contracts/SwapContract.sol#L226-L233
- https://github.com/SwingbyProtocol/skybridge-contract/blob/7f97b2aba97e704315c7b038d6a86d99964684c6/contracts/SwapContract.sol#L617
- https://github.com/SwingbyProtocol/skybridge-contract/blob/7f97b2aba97e704315c7b038d6a86d99964684c6/contracts/SwapContract.sol#L465
- https://github.com/SwingbyProtocol/skybridge-contract/blob/7f97b2aba97e704315c7b038d6a86d99964684c6/contracts/SwapContract.sol#L573

Wrong hierarchy

The **BurnableToken** contract inherits from an *IBurnableToken* interface, which is not necessary since in solidity, only contracts or abstract contracts are inherited, the interfaces are used as definitions, but it is not necessary for the contract to inherit from them, only that it implements these functionalities.

```
contract BurnableToken is Context, Ownable, IBurnableToken {
    using SafeMath for uint256;
```

Likewise, a series of methods with the override keyword have been detected in the same contract that are not necessary since they do not overwrite any virtual method of a parent class.

Source References

- https://github.com/SwingbyProtocol/skybridge-contract/blob/7f97b2aba97e704315c7b038d6a86d99964684c6/contracts/BurnableToken.sol#L9
- https://github.com/SwingbyProtocol/skybridge-contract/blob/7f97b2aba97e704315c7b038d6a86d99964684c6/contracts/BurnableToken.sol#L40-L83

Wrong Visibility

In order to simplify the contract for the users, it is recommended to turn the following variables to public.

```
uint256 private tokenTotalSupply;
string private tokenName;
string private tokenSymbol;
uint8 private tokenDecimals;
bool private isMintable;
```

When initializing variables as public the following methods will no longer be necessary, and can be removed:

```
/**
 * @dev Returns if the token is mintable or not
 */
function mintable() public override view returns (bool) {
    return isMintable;
}

/**
 * @dev Returns the token decimals.
 */
function decimals() public override view returns (uint8) {
    return tokenDecimals;
}

/**
 * @dev Returns the token symbol.
 */
function symbol() public override view returns (string memory) {
    return tokenSymbol;
}

/**
 * @dev Returns the token name.
 */
function name() public override view returns (string memory) {
    return tokenName;
}
```

Source References

- https://github.com/SwingbyProtocol/skybridge-contract/blob/7f97b2aba97e704315c7b038d6a86d99964684c6/contracts/BurnableToken.sol#L39-L64

Misleading names

The **SwapContract** contract implements the variables *lockedLPTokensForNode* and *feesLPTokensForNode*, it is recommended to modify its name to *lockedLPTokensForNodes* and *feesLPTokensForNodes* as plurals, since it is a cumulative for all nodes and its current name can cause confusion.

```
uint256 public lockedLPTokensForNode;
uint256 public feesLPTokensForNode;
```

We have the same example with the variable *lpDecimals*, which we consider that it should be called *lpConvertScale* to facilitate the interpretation of the real use of its value.

Source References

- https://github.com/SwingbyProtocol/skybridge-contract/blob/7f97b2aba97e704315c7b038d6a86d99964684c6/contracts/BurnableToken.sol#L39-L64
- https://github.com/SwingbyProtocol/skybridge-contract/blob/7f97b2aba97e704315c7b038d6a86d99964684c6/contracts/SwapContract.sol#L28

Lack of Event Index

Smart contract event indexing can be used to filter during event querying. This can be very useful when making dapps or in the off-chain processing of the events in our contract, as it allows filtering by specific addresses, making it much easier for developers to query the results of invocations.

It could be convenient to review the ***SwapContract*** contract to ensure that all the events have the necessary indexes for the correct functioning of the possible dApps. Addresses are usually the best argument to filter an event.

Source References

- https://github.com/SwingbyProtocol/skybridge-contract/blob/7f97b2aba97e704315c7b038d6a86d99964684c6/contracts/SwapContract.sol#L41-L66

Do not assume that ETH cannot be received

Although the logic of the contract itself does not imply any risk to the fact that it contains or not an amount of Ethereum, it is important to mention that the used protection to avoid receiving Ethereum will only prevent us from receiving a human error but never intentionally.

```
// The contract doesn't allow receiving Ether.
fallback() external {
    revert();
}
```

There are several ways to send ether on the Ethereum network without triggering the fallback method, such as, a contract destruction or by block rewards. So, we must assume that any ether or token sent to the contract cannot be withdrawn.

References

- https://medium.com/@aniketengg/ways-to-send-eth-to-a-contract-having-throw-in-fallback-function-41765db796de

Source References

- https://github.com/SwingbyProtocol/skybridge-contract/blob/7f97b2aba97e704315c7b038d6a86d99964684c6/contracts/SwapContract.sol#L705-L706
- https://github.com/SwingbyProtocol/skybridge-contract/blob/7f97b2aba97e704315c7b038d6a86d99964684c6/contracts/SwapContractFactory.sol#L29-L31

Invest in Security, invest in your future