SUPREMACY

Smart Contract Security Audit Report

Prepared for Bido Finance

Prepared by Supremacy

March 31, 2024

Contents

1 Introduction

Given the opportunity to review the design document and related codebase of the Bido Finance, we outline in the report our systematic approach to evaluate potential security issues in the smart contract(s) implementation, and provide additional suggestions or recommendations for improvement. Our results show that the given version of smart contracts can be further improved due to the presence of several issues related to either security or performance. This document outlines our audit results.

1.1 About Client

Bido is a liquid-staking protocol for BTC on the BEVM network, which is developed based on the lido on the Ethereum. Bido provides a simple way to obtain your digital token rewards. By staking with Bido, your tokens remain liquid and can be used in a range of DeFi applications to earn additional rewards.

Item	Description
Client	Bido Finance
Website	https://app.bido.finance
Type	Smart Contract
Languages	Solidity
Platform	EVM-compatible

1.2 Audit Scope

In the following, we show the Git repository of reviewed file and the commit hash used in this security audit:

- Repository: https://github.com/Bidohub/bido/tree/main/contracts
- Commit Hash: 871305d74404f09389fff6108ad95c5424841c61

Below are the files in scope for this security audit and their corresponding MD5 hashes.

Filename	MD5
`./Bido.sol`	`dac043a8b22a6ec3908997dab9ac9f1f`
`./StBTC.sol`	`17450d07b8616c7ac137a8a9d9997261`
`./WstBTC.sol`	`c2578da930eb74956375a8a18283112e`

1.3 Changelogs

Version	Date	Description
0.1	March 31, 2024	Initial Draft

1.4 About Us

Supremacy is a leading blockchain security firm, composed of industry hackers and academic researchers, provide top-notch security solutions through our technology precipitation and innovative research.

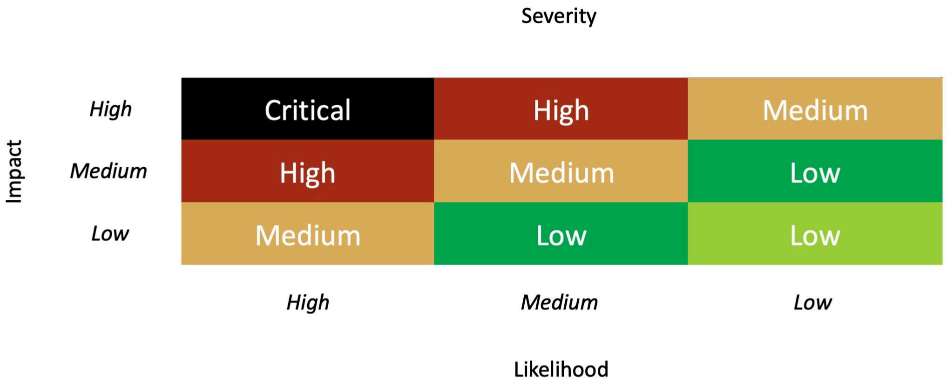
We are reachable at Twitter (https://twitter.com/SupremacyHQ), or Email (contact@supremacy.email).

1.5 Terminology

For the purpose of this assessment, we adopt the following terminology. To classify the severity of our findings, we determine the likelihood and impact (according to the CVSS risk rating methodology).

- Likelihood represents the likelihood of a finding to be triggered or exploited in practice
- Impact specifies the technical and business-related consequences of a finding
- Severity is derived based on the likelihood and the impact

We categorize the findings into four distinct categories, depending on their severity. These severities are derived from the likelihood and the impact using the following table, following a standard risk assessment procedure.



Severity

Impact	*High*	Critical	High	Medium
	Medium	High	Medium	Low
	Low	Medium	Low	Low
		High	*Medium*	*Low*
			Likelihood	

As seen in the table above, findings that have both a high likelihood and a high impact are classified as critical. Intuitively, such findings are likely to be triggered and cause significant disruption. Overall, the severity correlates with the associated risk. However, every finding's risk should always be closely checked, regardless of severity.

2 Findings

The table below summarizes the findings of the audit, including status and severity details.

ID	Severity	Description	Status
1	Informational	Refining code reuse with modifier	Undetermined
2	Informational	Optimizing code with ternary operators	Undetermined

2.1 Informational

1. Refining code reuse with modifier [Informational]

`Status: Undetermined`

Description:

In the `Bido::_stake()` and `Bido::_unstake()` function, the pause mechanism is used, which checks whether it is in a paused state or not by getting the return value of `BIDO_PAUSED_POSITION.getStorageBool()`. However, the code here is too repetitive.

```
    /**
     * @dev Process user stake, mints liquid tokens
     * @param _referral address of referral.
     * @return amount of StBTC shares generated
     */
    function _stake(address _referral) internal returns (uint256) {
        require(_sharesOf(INITIAL_TOKEN_HOLDER) != 0, "NOT_INITIALIZED");
        require(msg.value != 0, "ZERO_DEPOSIT");

        require(!BIDO_PAUSED_POSITION.getStorageBool(), "STAKING_PAUSED");

        uint256 totalPooledBtc = _getTotalPooledBtc();
        uint256 preTotalPooledBtc = totalPooledBtc.sub(msg.value);

        uint256 sharesAmount = msg.value.mul(_getTotalShares()).div(
            preTotalPooledBtc
        );

        _mintShares(msg.sender, sharesAmount);

        emit Staked(msg.sender, msg.value, _referral);

        _emitTransferAfterMintingShares(msg.sender, sharesAmount);
        return sharesAmount;
    }
```

Bido.sol

Recommendation: Could be solved by wrapping a modifier or revise the visibility of the `isStakingPaused()` function so that it can be reused.

2. Optimizing code with ternary operators [Informational]

Status: Undetermined

Description:

In the `Bido::_unstake()` function, the conditional expressions that get `sharesAmount` values can be optimized.

```
    /**
     * @dev Process user unstake, burns liquid tokens
     */
    function _unstake(uint256 _amount) internal {
        require(!BIDO_PAUSED_POSITION.getStorageBool(), "STAKING_PAUSED");
        require(_amount != 0, "UNSTAKE_ZERO");

        uint256 sharesAmount;
        if (_amount == MAX_INT) {
            sharesAmount = _sharesOf(msg.sender);
        } else {
            sharesAmount = getSharesByPooledBtc(_amount);
        }

        require(sharesAmount != 0, "BURN_ZERO");

        (, uint256 preRebaseTokenAmount) = _burnShares(
            msg.sender,
            sharesAmount
        );

        _sendValue(msg.sender, preRebaseTokenAmount);

        emit UnStaked(msg.sender, preRebaseTokenAmount);
    }
```

Bido.sol

Recommendation: Revise the code logic for ternary operators.

```
    /**
     * @dev Process user unstake, burns liquid tokens
     */
    function _unstake(uint256 _amount) internal {
        require(!BIDO_PAUSED_POSITION.getStorageBool(), "STAKING_PAUSED");
        require(_amount != 0, "UNSTAKE_ZERO");

        uint256 sharesAmount = _amount > _sharesOf(msg.sender) ? _sharesOf(msg.sender) : getSharesByPooledBtc(_amount);

        require(sharesAmount != 0, "BURN_ZERO");

        (, uint256 preRebaseTokenAmount) = _burnShares(
            msg.sender,
            sharesAmount
        );

        _sendValue(msg.sender, preRebaseTokenAmount);

        emit UnStaked(msg.sender, preRebaseTokenAmount);
    }
```

Bido.sol

3 Disclaimer

This security audit report does not constitute investment advice or a personal recommendation. It 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. Any entity should not 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. This security audit report is not an endorsement of any particular project or team, and the report does not guarantee the security of any particular project. This audit does not give any warranties on discovering all security issues of the blockchain module, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues, also cannot make guarantees about any additional code added to the assessed project after the audit version. As one audit-based assessment cannot be considered comprehensive, we always recommend proceeding with independent audits and a public bug bounty program to ensure the security of blockchain(s) module. Unless explicitly specified, the security of the language itself (e.g., the solidity language), the underlying compiling toolchain and the computing infrastructure are out of the scope.