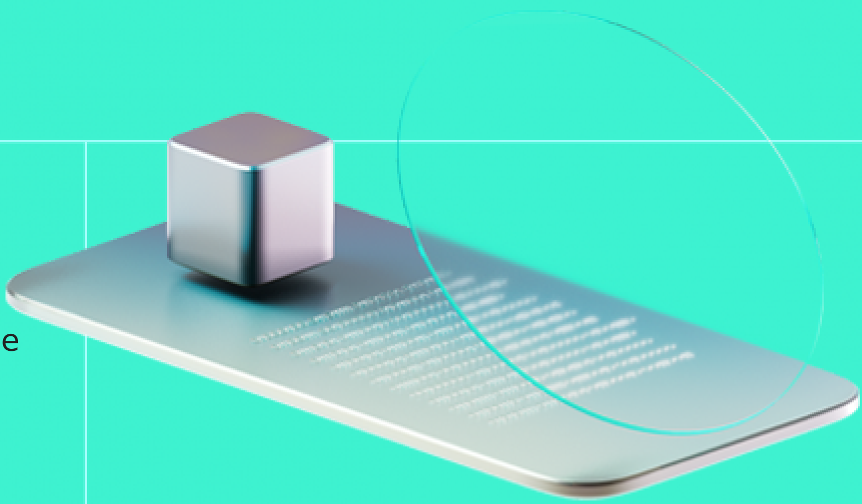
Smart Contract Code Review And Security Analysis Report



Customer: Byzantine Finance

Date: 03/02/2025

We express our gratitude to the Byzantine Finance team for the collaborative engagement that enabled the execution of this Smart Contract Security Assessment.

Byzantine is a trustless and restaking layer with permissionless strategy creation. Byzantine enables the deployment of minimal, individual, and isolated restaking strategy vaults.

Document

Name	Smart Contract Code Review and Security Analysis Report for Byzantine Finance
Audited By	Kornel Światłowski
Approved By	Grzegorz Trawinski
Website	https://www.byzantine.fi/
Changelog	28/01/2025 - Preliminary Report; 03/02/2025 - Final Report
Platform	Ethereum
Language	Solidity
Tags	Vault
Methodology	https://hackenio.cc/sc_methodology

Review Scope

Repository	https://github.com/Byzantine-Finance/predeposit-contract-ethereum
Commit	2d26ab7708bac5cfce666fd34990da055d0bc9ac
Retest Commit	a48a87e85c566548736cb0d4d01b90967783600f

Audit Summary

The system users should acknowledge all the risks summed up in the risks section of the report

1	1	0	0
Total Findings	Resolved	Accepted	Mitigated

Findings by Severity

Severity	Count
Critical	0
High	0
Medium	0
Low	1

Vulnerability	Severity
F-2025-8497 - Missing Zero Address Validation Can Lead To Token Lock	Low

Documentation quality

- Functional requirements are detailed.
- Technical description is detailed.

Code quality

- The development environment is configured.

Test coverage

Code coverage of the project is **69.77%** (branch coverage).

- Deployment and basic user interactions are covered with tests.
- Fuzzing tests are provided.
- Negative cases coverage is missed.
- Not every function is tested.

Table of Contents

System Overview

The `ByzantineDeposit` contract facilitates deposits of whitelisted tokens, including `ETH`, `stETH`, and `wstETH` by default, into the Byzantine protocol. It supports both `ERC20` and `ETH` deposits, with restrictions for authorized addresses unless permissionless deposits are enabled. Deposited `stETH` tokens are automatically wrapped into `wstETH` to prevent balance fluctuations due to rebasing. The contract also enables transferring deposited assets to approved Byzantine vaults while enforcing asset compatibility and share allocation constraints.

The `PauserRegistry` contract manages the roles of pausers and a single unpauser, enabling role-based control mechanisms. The unpauser role, designed to be more secure, holds exclusive authority to assign or revoke pauser roles and update the unpauser address itself. This design ensures centralized governance of pausing functionality, with safeguards against unauthorized changes through role verification.

The `Pausable` contract implements a modular pausability mechanism, allowing controlled suspension of functionality based on bitwise flags. Access control for pausing and unpausing is delegated to a `PauserRegistry` contract, where authorized roles such as pauser and unpauser manage the state transitions. The design supports granular control of functionality by enabling or disabling specific features through individual bits in the `_paused` state variable, ensuring flexible and secure operations.

Privileged roles

The `ByzantineDeposit` contract uses a `Ownable2Step` library from OpenZeppelin to restrict access to important functions.

Contract owner can:

- Sets whether some addresses are authorized to deposit.
- Adds a new token to the list of allowed deposit tokens.
- Remove a token from the list of allowed deposit tokens.
- Change the permissionless deposit status.
- Record Byzantine Vaults to allow moving tokens to them.
- Delist a Byzantine Vault.

The `PauserRegistry` and `Pausable` contracts use a custom role-based control mechanism. Roles assigned in the `PauserRegistry` contract are used as well in the `Pausable` contract. Contract defines two roles, pauser (multiple addresses) and unpasuer (single address) role.

Unpauser role can:

- Sets new unpauser.
- Sets new pauser.
- Unpauses a Byzantine contract's functionality.
- Sets a new pauser registry.

Pauser role can:

- Pauses a Byzantine contract's functionality.

Potential Risks

- System Reliance on External Contracts: The `moveToVault()` function significantly relies on specific external contracts. Any flaws or vulnerabilities in these contracts adversely affect the audited project, potentially leading to security breaches or loss of funds.
- Owner's Unrestricted State Modification: The absence of restrictions on state variable modifications by the owner leads to arbitrary changes, affecting contract integrity and user trust.
- The `ByzantineDeposit` contract allows the owner to add new tokens for deposits via the `addDepositToken()` function. Per the documentation and description of the `addDepositToken()` , the owner is responsible for avoiding the addition of "exotic ERC20 tokens." The `depositERC20()` function is incompatible with fee-on-transfer tokens, as it records the full transfer amount, including the fee, leading to incorrect functionality.

Findings

Vulnerability Details

F-2025-8497 - Missing Zero Address Validation Can Lead To Token Lock - Low

Description:

In Solidity, the Ethereum address `0x0000000000000000000000000000000000000000` is known as the **zero address**. This address has significance because it is the default value for uninitialized address variables and is often used to represent an invalid or non-existent address.

The issue arises when a Solidity smart contract does not properly check or prevent interactions with the zero address, leading to unintended behavior. For instance, a contract might allow tokens to be sent to the zero address without any checks, which essentially burns those tokens as they become irretrievable. While sometimes this is intentional, without proper control or checks, accidental transfers could occur.

The `_receiver` address input parameter in the `withdraw()` and `moveToVault()` functions lacks proper validation against the zero address:

- `withdraw()`: When withdrawing ETH, the absence of validation could result in ETH being sent to the zero address, leading to locked funds. For ERC20 token withdrawals, validation is performed internally by the ERC20 contract.
- `moveToVault()`: The implementation of the `IERC7535` and `IERC4626` `deposit()` function is external and unknown, meaning passing the zero address as `_receiver` may result in locked tokens that cannot be accessed.

```
function withdraw(IERC20 _token, uint256 _amount, address _receiver) external
nonReentrant {
    if (depositedAmount[msg.sender][_token] < _amount)
        revert InsufficientDepositedBalance(msg.sender, _token);
    unchecked {
        // Overflow not possible because of previous check
        depositedAmount[msg.sender][_token] -= _amount;
    }

    if (_token == beaconChainETHToken) {
        (bool success, ) = _receiver.call{value : _amount}("");
```

```
        if (!success) revert ETHTransferFailed();
        emit Withdraw(msg.sender, _token, _amount, _receiver);
        return;
    } else if (_token == stETHToken) {
        _amount = wstETH.unwrap(_amount);
    }
    _token.safeTransfer(_receiver, _amount);
    emit Withdraw(msg.sender, _token, _amount, _receiver);
}

function moveToVault(IERC20 _token, address _vault, uint256 _amount,
 address _receiver,
                     uint256 _minSharesOut) external
    onlyWhenNotPaused(PAUSED_VAULTS_MOVES) nonReentrant {
    require(canDeposit[msg.sender],
            "ByzantineDeposit.moveToVault: address is not authorized to move
tokens");
    if (!isByzantineVault[_vault]) revert NotAllowedVault(_vault);
    if (address(_token) != IERC4626(_vault).asset()) revert MismatchingAssets
();
    if (depositedAmount[msg.sender][_token] < _amount)
        revert InsufficientDepositedBalance(msg.sender, _token);
    unchecked {
        // Overflow not possible because of previous check
        depositedAmount[msg.sender][_token] -= _amount;
    }

    uint256 sharesBefore;
    uint256 sharesAfter;
    if (_token == beaconChainETHToken) {
        sharesBefore = IERC7535(_vault).balanceOf(_receiver);
        IERC7535(_vault).deposit{value : _amount}(_amount, _receiver);
        sharesAfter = IERC7535(_vault).balanceOf(_receiver);
    } else {
        if (_token == stETHToken) {
            _amount = wstETH.unwrap(_amount);
        }

        _token.forceApprove(_vault, _amount);
        sharesBefore = IERC4626(_vault).balanceOf(_receiver);
        IERC4626(_vault).deposit(_amount, _receiver);
        sharesAfter = IERC4626(_vault).balanceOf(_receiver);
    }

    uint256 sharesReceived = sharesAfter - sharesBefore;
    if (sharesReceived < _minSharesOut) revert InsufficientSharesReceived();
    emit MoveToVault(msg.sender, _token, _vault, _amount, _receiver);
}
```

Assets:

- src/ByzantineDeposit.sol [https://github.com/Byzantine-Finance/predeposit-contract-ethereum]

Status: Fixed

Classification

Impact: 3/5

Likelihood: 2/5

Exploitability: Independent

Complexity: Simple

Severity: Low

Recommendations

Remediation: It is recommended to implement explicit validation checks for the `_receiver` address in the `withdraw()` and `moveToVault()` functions to ensure it is not the zero address. This validation would prevent unintended transfers to the zero address and mitigate the risk of asset loss.

Resolution: The Finding was fixed in commit `ac5cbe7b0591ea5c801f742cb8487ecdd1bd8773`. The validation against the zero address for the `_receiver` address was added to the `withdraw()` and `moveToVault()` functions.

Observation Details

F-2025-8484 - Floating Pragma - Info

Description: In Solidity development, the pragma directive specifies the compiler version to be used, ensuring consistent compilation and reducing the risk of issues caused by version changes. However, using a floating pragma introduces uncertainty, as it allows contracts to be compiled with any version within a specified range. This can result in discrepancies between the compiler used in testing and the one used in deployment, increasing the likelihood of vulnerabilities or unexpected behavior due to changes in compiler versions.

The project currently uses floating pragma declarations `^0.8.20` in its Solidity contracts. This increases the risk of deploying with a compiler version different from the one tested, potentially reintroducing known bugs from older versions or causing unexpected behavior with newer versions. These inconsistencies could result in security vulnerabilities, system instability, or financial loss. Locking the pragma version to a specific, tested version is essential to prevent these risks and ensure consistent contract behavior.

Assets:

- src/ByzantineDeposit.sol [https://github.com/Byzantine-Finance/predeposit-contract-ethereum]
- src/permissions/Pausable.sol [https://github.com/Byzantine-Finance/predeposit-contract-ethereum]
- src/permissions/PauserRegistry.sol [https://github.com/Byzantine-Finance/predeposit-contract-ethereum]

Status: Fixed

Recommendations

Remediation: It is recommended to lock the pragma version to the specific version that was used during development and testing. This ensures that the contract will always be compiled with a known, stable compiler version, preventing unexpected changes in behavior due to compiler updates. For example, instead of using `^0.8.xx`, explicitly define the version with `pragma solidity 0.8.19;`.

Resolution: The Finding was fixed in commit `ac5cbe7b0591ea5c801f742cb8487ecdd1bd8773`. The pragma was locked to version `0.8.28`.

F-2025-8485 - Violation of the Checks-Effects-Interactions (CEI) pattern - Info

Description:

The Checks-Effects-Interactions (CEI) pattern is a widely accepted best practice in smart contract development, designed to mitigate the risk of re-entrancy attacks. This pattern mandates that contracts should first perform checks, such as `require` statements, then execute effects, such as state changes, and finally carry out any interactions with external contracts. Failing to follow this pattern can expose the contract to re-entrancy attacks, where an attacker can repeatedly call a vulnerable function before its state is updated, potentially draining contract assets.

In the `depositERC20()` function of the `ByzantineDeposit` contract, the CEI pattern is not consistently followed. The `depositedAmount` variable is updated after interactions with the `stETHToken` and `wstETH` contracts, which violates the CEI pattern. Full adherence to the CEI pattern is not feasible in this case, as the update of `depositedAmount` relies on the return value of the `wstETH.wrap()` function call.

```
function depositERC20(
    IERC20 _token,
    uint256 _amount
) external onlyWhenNotPaused(PAUSED_DEPOSITS) onlyIfCanDeposit(msg.sender) {
    if (!isDepositToken[_token]) revert NotAllowedDepositToken(_token);
    _token.safeTransferFrom(msg.sender, address(this), _amount);
    uint256 amount = _amount;
    if (_token == stETHToken) {
        stETHToken.approve(address(wstETH), _amount);
        amount = wstETH.wrap(_amount);
    }
    depositedAmount[msg.sender][_token] += amount;
    emit Deposit(msg.sender, _token, _amount);
}
```

Assets:

- src/ByzantineDeposit.sol [https://github.com/Byzantine-Finance/predeposit-contract-ethereum]

Status: Accepted

Recommendations

Remediation: It is recommended to apply the `nonReentrant` modifier from the already imported `ReentrancyGuard` library to the `depositERC20()` function. This will mitigate the risk of re-entrancy attacks by ensuring that the function cannot be re-entered during its execution, even if the CEI pattern cannot be strictly followed.

Resolution: The Finding was accepted by the Client.

F-2025-8514 - Single-Step Role Transfer in PauserRegistry Allows Permanent Control Loss - Info

Description:

The `PauserRegistry` contract implements a custom role-based access control mechanism with two defined roles: `pauser` and `unpauser`. Multiple addresses can hold the `pauser` role, but only one address can hold the `unpauser` role at any given time. The `unpauser` role has elevated privileges, including the ability to add or revoke addresses from the `pauser` role.

The `unpauser` role can also be transferred to another address. However, the role transfer mechanism follows a single-step process, allowing the immediate transfer of this role to a new address. The `unpauser` role could be transferred to an invalid or incorrect address (e.g., due to a typo), including the zero address. If the `unpauser` role is transferred to an address that cannot manage it, control over critical contract functions could be permanently lost.

```
function setUnpauser(address newUnpauser) external onlyUnpauser {
    _setUnpauser(newUnpauser);
}

function _setUnpauser(address newUnpauser) internal {
    require(newUnpauser != address(0), "PauserRegistry._setUnpauser: zero add
ress input");
    emit UnpauserChanged(unpauser, newUnpauser);
    unpauser = newUnpauser;
}
```

Assets:

- src/permissions/PauserRegistry.sol [https://github.com/Byzantine-Finance/predeposit-contract-ethereum]

Status: Accepted

Recommendations

Remediation: It is recommended to implement a two-step transfer mechanism for the `unpauser` role. This mechanism should involve an initiation step where the current `unpauser` designates a new address, followed by a confirmation step where the designated address explicitly accepts the role.

Resolution: The Finding was accepted with the following statement:

It is very unlikely that this issue occurs… The unpauser will be a safe address with a high threshold. Before signing the tx, every person in the safe will double check the new unpauser address.

F-2025-8525 - Redundant Imports - Info

Description:

The `ByzantineDeposit` contract includes separate imports of `Ownable` and `IERC20`, despite more efficient alternatives being available through consolidated imports. Specifically:

- The `Ownable` contract is imported directly but can be accessed through `Ownable2Step`, which already includes the `Ownable`.
- The `IERC20` interface is imported directly but can be accessed through `SafeERC20`, which already includes the `IERC20` interface.

These redundant imports reduce overall code quality.

```
import {SafeERC20} from "@openzeppelin/contracts/token/ERC20/utils/SafeERC20.
sol";
import {Ownable2Step} from "@openzeppelin/contracts/access/Ownable2Step.sol";
import {Ownable} from "@openzeppelin/contracts/access/Ownable.sol";
import {IERC20} from "@openzeppelin/contracts/token/ERC20/IERC20.sol";
```

Assets:

- src/ByzantineDeposit.sol [https://github.com/Byzantine-Finance/predeposit-contract-ethereum]

Status: Fixed

Recommendations

Remediation:

It is recommended to consolidate imports by accessing the `Ownable` through `Ownable2Step` and `IERC20` interface through `SafeERC20`. This approach reduces redundancy, improves code maintainability. For example, the updated imports can be structured as:

```
import {SafeERC20, IERC20} from "@openzeppelin/contracts/token/ERC20/utils/Sa
feERC20.sol";
import {Ownable2Step, Ownable} from "@openzeppelin/contracts/access/Ownable2S
tep.sol";
```

Resolution:

The Finding was fixed in commit `ac5cbe7b0591ea5c801f742cb8487ecdd1bd8773`. The imports were consolidated. `IERC20` is imported from `SafeERC20`, and `Ownable` is imported from `Ownable2Step`.

F-2025-8528 - Unchecked Return Value of approve() Function - Info

Description:

The `depositERC20` function currently lacks a step in its implementation by not verifying the return value of the call to the `approve()` function. This omission introduces a potential risk, as the success or failure of the approval transaction is not being validated, which can lead to unforeseen issues and jeopardize the intended functionality of the contract.

```
function depositERC20(IERC20 _token, uint256 _amount) external onlyWhenNotPau
sed(PAUSED_DEPOSITS)
    onlyIfCanDeposit(msg.sender) {
    if (!isDepositToken[_token]) revert NotAllowedDepositToken(_token);
    _token.safeTransferFrom(msg.sender, address(this), _amount);
    uint256 amount = _amount;
    if (_token == stETHToken) {
        stETHToken.approve(address(wstETH), _amount);
        amount = wstETH.wrap(_amount);
    }
    depositedAmount[msg.sender][_token] += amount;
    emit Deposit(msg.sender, _token, _amount);
}
```

Assets:

- src/ByzantineDeposit.sol [https://github.com/Byzantine-Finance/predeposit-contract-ethereum]

Status: Fixed

Recommendations

Remediation: It is suggested to use OpenZeppelin's `forceApprove()` functions from the `SafeERC20` library similar to the `moveToVault()` function to validate the approve function's return value.

Resolution: The Finding was fixed in commit `ac5cbe7b0591ea5c801f742cb8487ecdd1bd8773`. The `approve()` was replaced with `forceApprove()` in the `depositERC20()` function, ensuring proper handling of the return value.

Disclaimers

Hacken Disclaimer

The smart contracts given for audit have been analyzed based on best industry practices at the time of the writing of this report, with cybersecurity vulnerabilities and issues in smart contract source code, the details of which are disclosed in this report (Source Code); the Source Code compilation, deployment, and functionality (performing the intended functions).

The report contains no statements or warranties on the identification of all vulnerabilities and security of the code. The report covers the code submitted and reviewed, so it may not be relevant after any modifications. Do not consider this report as a final and sufficient assessment regarding the utility and safety of the code, bug-free status, or any other contract statements.

While we have done our best in conducting the analysis and producing this report, it is important to note that you should not rely on this report only — we recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contracts.

English is the original language of the report. The Consultant is not responsible for the correctness of the translated versions.

Technical Disclaimer

Smart contracts are deployed and executed on a blockchain platform. The platform, its programming language, and other software related to the smart contract can have vulnerabilities that can lead to hacks. Thus, the Consultant cannot guarantee the explicit security of the audited smart contracts.

Appendix 1. Definitions

Severities

When auditing smart contracts, Hacken is using a risk-based approach that considers **Likelihood**, **Impact**, **Exploitability** and **Complexity** metrics to evaluate findings and score severities.

Reference on how risk scoring is done is available through the repository in our Github organization:

hknio/severity-formula

Severity	Description
Critical	Critical vulnerabilities are usually straightforward to exploit and can lead to the loss of user funds or contract state manipulation.
High	High vulnerabilities are usually harder to exploit, requiring specific conditions, or have a more limited scope, but can still lead to the loss of user funds or contract state manipulation.
Medium	Medium vulnerabilities are usually limited to state manipulations and, in most cases, cannot lead to asset loss. Contradictions and requirements violations. Major deviations from best practices are also in this category.
Low	Major deviations from best practices or major Gas inefficiency. These issues will not have a significant impact on code execution.

Potential Risks

The "Potential Risks" section identifies issues that are not direct security vulnerabilities but could still affect the project’s performance, reliability, or user trust. These risks arise from design choices, architectural decisions, or operational practices that, while not immediately exploitable, may lead to problems under certain conditions. Additionally, potential risks can impact the quality of the audit itself, as they may involve external factors or components beyond the scope of the audit, leading to incomplete assessments or oversight of key areas. This section aims to provide a broader perspective on factors that could affect the project's long-term security, functionality, and the comprehensiveness of the audit findings.

Appendix 2. Scope

The scope of the project includes the following smart contracts from the provided repository:

Scope Details	
Repository	https://github.com/Byzantine-Finance/predeposit-contract-ethereum
Commit	2d26ab7708bac5cfce666fd34990da055d0bc9ac
Retest Commit	a48a87e85c566548736cb0d4d01b90967783600f
Whitepaper	https://docs.byzantine.fi/
Requirements	README.md
Technical Requirements	README.md

Asset	Type
src/ByzantineDeposit.sol [https://github.com/Byzantine-Finance/predeposit-contract-ethereum]	Smart Contract
src/permissions/Pausable.sol [https://github.com/Byzantine-Finance/predeposit-contract-ethereum]	Smart Contract
src/permissions/PauserRegistry.sol [https://github.com/Byzantine-Finance/predeposit-contract-ethereum]	Smart Contract

Appendix 3. Additional Valuables

Verification of System Invariants

During the audit of **Byzantine Finance / Deposit**, Hacken followed its methodology by performing fuzz-testing on the project's main functions. Foundry, a tool used for fuzz-testing, was employed to check how the protocol behaves under various inputs. Due to the complex and dynamic interactions within the protocol, unexpected edge cases might arise. Therefore, it was important to use fuzz-testing to ensure that several system invariants hold true in all situations.

Fuzz-testing allows the input of many random data points into the system, helping to identify issues that regular testing might miss. A specific Echidna fuzzing suite was prepared for this task, and throughout the assessment, 5 invariants were tested over 1,000,000 runs each. This thorough testing ensured that the system works correctly even with unexpected or unusual inputs.

Invariant	Test Result	Run Count
Deposit random amount of ETH	Passed	1,000,000
Deposit random amount of ETH by two users	Passed	1,000,000
Withdraw random amount of ETH	Passed	1,000,000
Deposit random amount of wstETH	Passed	1,000,000
Withdraw random amount of wstETH	Passed	1,000,000

Additional Recommendations

The smart contracts in the scope of this audit could benefit from the introduction of automatic emergency actions for critical activities, such as unauthorized operations like ownership changes or proxy upgrades, as well as unexpected fund manipulations, including large withdrawals or minting events. Adding such mechanisms would enable the protocol to react automatically to unusual activity, ensuring that the contract remains secure and functions as intended.

To improve functionality, these emergency actions could be designed to trigger under specific conditions, such as:

- Detecting changes to ownership or critical permissions.
- Monitoring large or unexpected transactions and minting events.
- Pausing operations when irregularities are identified.

These enhancements would provide an added layer of security, making the contract more robust and better equipped to handle unexpected situations while maintaining smooth operations.