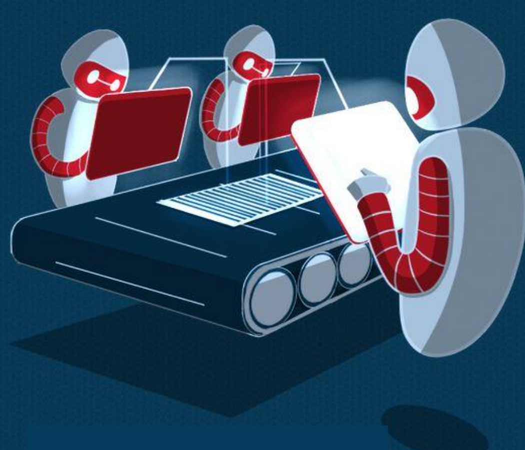
SMART CONTRACT SECURITY ANALYSIS REPORT

```
pragma solidity 0.7.0;
contract Contract {

     function hello() public returns (string) {
         return "Hello World!";
     }

     function findVulnerability() public returns (string) {
         return "Finding Vulnerability";
     }

     function solveVulnerability() public returns (string) {
         return "Solve Vulnerability";
     }
}
```



Objectives

The purpose of this document is to highlight any identified bugs/issues in the provided codebase. This security review has been conducted in a closed and secure environment, free from influence or bias of any sort. This document may contain confidential information about the IT system/ architecture and the intellectual property of the client. It also may contain information about potential risks and the processes involved in mitigating/ exploiting the identified risks. The usage of information provided in this report is limited, internally, to the client. However, this report can be disclosed publicly with the intention to aid our growing blockchain community; only at the discretion of the client.

Key understandings

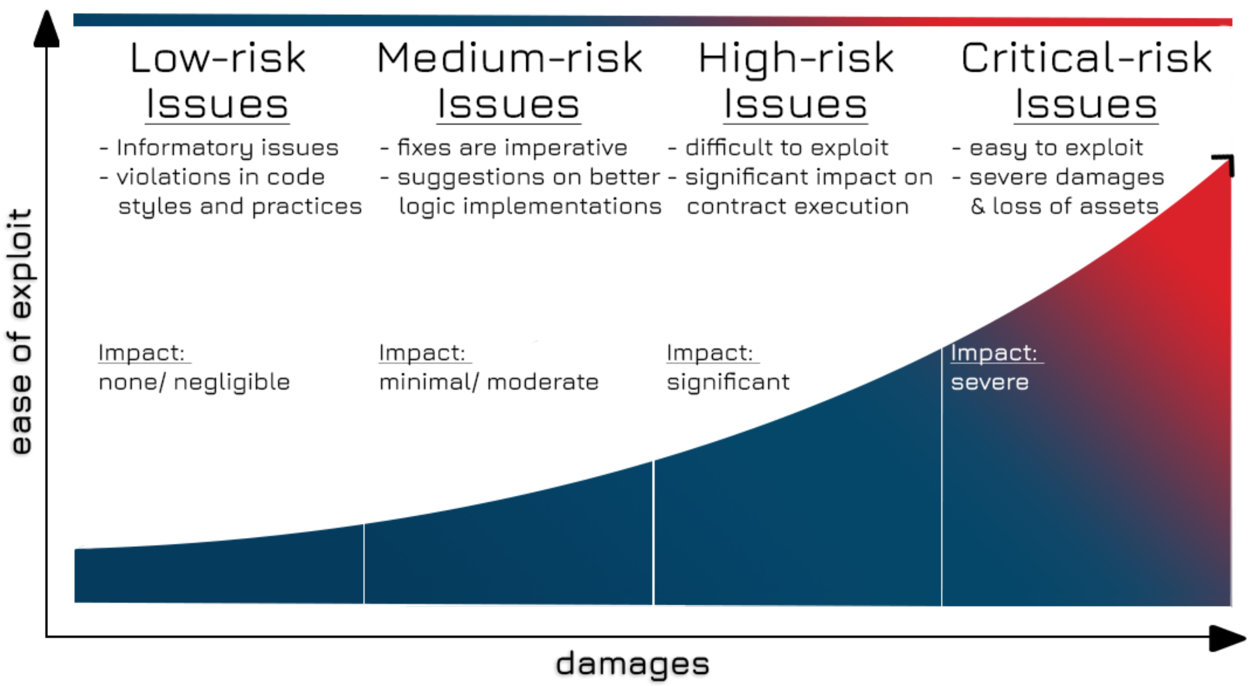


TABLE OF CONTENTS

INTRODUCTION

BlockApex (Security Reviewer) was contracted by BaseYield (Client) for the purpose of conducting a Smart Contract Code Security Review. This document presents the findings of our analysis which started on 24th Nov '2023

Name
BaseYield
Security Researchers
Kaif Ahmed | Muhammad Jariruddin
Platform
Ethereum | EVM Compatible Blockchains | Solidity
Type of review
Manual Code Review | Automated Tools Analysis
Methods
Architecture Review | Functional Testing | Computer-Aided Verification | Manual Review
Git repository/ Commit Hash
Repo | 538f853dba67d2eb657e65c27e456062eeb1d442
White paper/ Documentation
-
Document log
Initial Review Completed: Nov 28 '2023
Final Review: Dec 4 '2023

Scope

The shared git-repository/ codebase was checked for common code violations along with vulnerability-specific probing to detect major issues/vulnerabilities.

Some specific attack vectors and threat surfaces are as follows:

Code review		Functional review
Reentrancy	Unchecked external call	Business Logics Review
Ownership Takeover	Fungible token violations	Functionality Checks
Timestamp Dependence	Unchecked math	Access Control & Authorization
Gas Limit and Loops	Unsafe type inference	Escrow manipulation
DoS with (Unexpected) Throw	Implicit visibility level	Token Supply manipulation
DoS with Block Gas Limit	Deployment Consistency	Asset's integrity
Transaction-Ordering Dependence	Repository Consistency	User Balances manipulation
Style guide violation	Data Consistency	Kill-Switch Mechanism
Costly Loop		Operation Trails & Event Generation

Project Overview

BayVault is a yield-optimizing vault contract that interfaces with users for deposits and withdrawals. It inherits from ERC20Upgradeable, OwnableUpgradeable, and ReentrancyGuardUpgradeable, utilizing OpenZeppelin contracts for standard functionalities. The contract is designed to work in tandem with an external strategy, defined by the IStrategyV7 interface.

Strategy Management

The contract maintains a current strategy for yield optimization and allows for the proposal of new strategies through the StratCandidate structure. An approval delay mechanism is in place to ensure the security and integrity of strategy transitions.

Deposit and Withdrawal Mechanisms

Users can deposit assets into the vault, which mints corresponding vault tokens. Deposited assets are then transferred to the strategy for yield generation. Withdrawals burn vault tokens and return the user's share of the underlying assets, retrieving funds from the strategy if necessary.

Token and Share Value Management

The contract handles an underlying asset (referred to as want) and issues its own vault tokens as shares. It includes a function to calculate the price per share, providing transparency on the value of holdings within the vault.

System Interaction and Flow

BayVault operates as a gateway for users to deposit assets for yield optimization, interacting seamlessly with an external strategy while providing a secure and transparent environment for asset management.

Considerations and Assumptions:

- The contracts have mechanisms to ensure the validity of token addresses and prices and the availability of sufficient deposits for user operations.
- The system seems to be designed with security in mind, implementing safeguards like ReentrancyGuard and ensuring proper access control with modifiers like onlyOwner.
- The contracts are well-structured and modular, allowing for a clear separation of concerns and responsibilities among different components of the system.

System Architecture

Contract Structure

- **Inheritance**: BayVault inherits from ERC20Upgradeable, OwnableUpgradeable, and ReentrancyGuardUpgradeable, leveraging OpenZeppelin's robust and secure implementations.
- **Components**: The contract comprises core functionalities for token handling (deposit/withdrawal), strategy management, and emergency procedures.

User Interaction Layer

- **Deposit and Withdrawal Functions**: Users interact with these functions to move assets into and out of the vault. The contract mints and burns vault tokens to represent the user's share.
- **Token Management**: Handles the "want" token, ensuring that user deposits and withdrawals are processed accurately.

Strategy Integration

- **External Strategy Contract**: BayVault delegates yield optimization to an external strategy defined by the IStrategyV7 interface.
- **Dynamic Interaction**: The vault actively communicates with the strategy for depositing funds and managing withdrawals, maintaining an efficient flow of assets.

Governance and Upgradeability

- **Strategy Management**: The contract owner can propose and upgrade strategies, with an approval delay to ensure security and stability.
- **Owner-Only Emergency Functions**: Functions like inCaseTokensGetStuck provide a safeguard against unforeseen scenarios, allowing the owner to retrieve non-strategy tokens.

Security Mechanisms

- **Reentrancy Guard**: Protects against reentrancy attacks, a critical feature for contracts handling external calls and transfers.
- **Ownership Controls**: Ensures that sensitive actions, like strategy upgrades, can only be executed by the contract owner.

Transparency and Auditability

- **Event Logging**: Key actions within the contract emit events, providing transparency and facilitating off-chain monitoring and auditing.

Conclusion

The BayVault smart contract presents a robust and efficient system for yield optimization in the DeFi space. It is designed to facilitate user interactions for depositing and withdrawing assets, while an external strategy contract handles the complexities of yield generation. The system's architecture ensures that user funds are actively put to work in the strategy for optimal returns while also providing a mechanism for safe and proportional withdrawals.

Key features like the minting and burning vault tokens align users' interests with the vault's performance, creating a transparent and fair system for tracking individual shares. The contract's integration with a strategy diversifies its functionality and allows for adaptability and upgrades in response to the evolving DeFi landscape.

Methodology & Scope

The codebase went through a security review using a filtered code review technique. A pair of two (2) security researchers scanned the codebase in an iterative process for two (2) weeks.

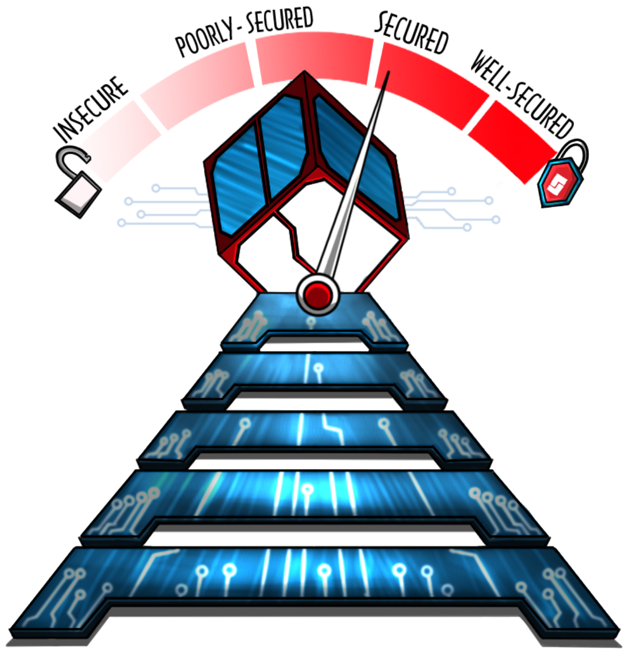
1. The Security Review started with the reconnaissance phase, and a basic understanding was developed.
2. The security researchers worked on developing presumptions for the production-ready codebase and the client protocol's relevant documentation/ white paper.
3. The security audit moved up to the manual code reviews to find logical flaws in the codebase.
4. Further complemented with code optimizations, software, and security design patterns implementation, code styles, best practices, and identifying false positives detected by automated analysis tools.
5. The auditors only took the **bayVault.sol** contract in the scope, any external calls sent to strategy are assumed to be safe and out-of-scope of this audit.

SECURITY REVIEW REPORT

Executive Summary

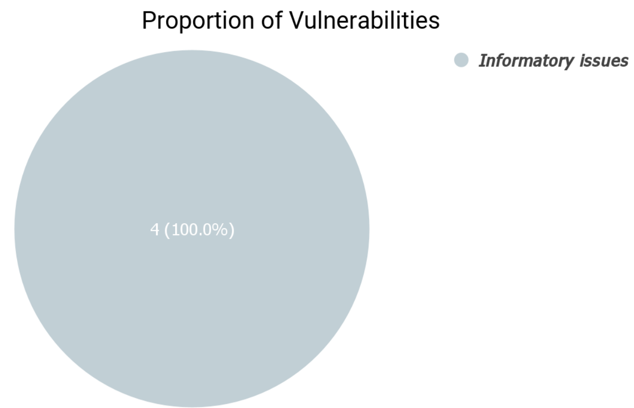
Our team performed a technique called "Filtered Security Review," where two individuals reviewed the BayVault smart contract separately.

After a thorough and rigorous process involving manual code review, automated testing was carried out using; Slither for static analysis All the flags raised were manually reviewed and re-tested to identify the false positives.



Our team found:

Issues	Severity Level	Open	Resolved	Acked
-	Critical	-	-	-
-	High	-	-	-
-	Medium	-	-	-
-	Low	-	-	-
4	Informatory	-	-	4
-	Undetermined	-	-	-



Key Findings

#	Findings	Risk	Status
1.	Inadequate Withdrawal Logic Handling	Informatory	Acknowledged
2.	Suboptimal Function Organization	Informatory	Acknowledged
3.	Non-Intuitive Variable Naming	Informatory	Acknowledged
4.	Inefficient Use of External Function Visibility	Informatory	Acknowledged

Findings

Detailed Overview

Informational and Gas optimisation issues

ID	1
Title	Inadequate Withdrawal Logic Handling
Path	contracts/BayVault.sol
Function Name	withdraw( )

Description: The withdrawal logic in the contract does not account for scenarios where the vault has insufficient funds because they are deployed in the strategy. The current logic assumes that the required funds are always available in the vault, which is only sometimes the case. This oversight can lead to requests for withdrawals that cannot be fulfilled, or the calculations for withdrawal amounts become irrelevant.

Impact: The flawed withdrawal logic can lead to significant user experience issues. If the strategy needs to be invoked frequently to fulfill withdrawals, there is no need to check for the before and after balance assuming that there will be no funds in the contract, placing such calculations and checks could lead to increased transaction costs and potential delays, further degrading user experience.

Code Reference:

```
function withdraw(uint256 _shares) public {
    uint256 r = (balance() * _shares) / totalSupply();
    _burn(msg.sender, _shares);

    uint b = want().balanceOf(address(this));
    if (b < r) {
        uint _withdraw = r - b;
```

```
            strategy.withdraw(_withdraw);
            uint _after = want().balanceOf(address(this));
            uint _diff = _after - b;
            if (_diff < _withdraw) {
                r = b + _diff;
            }
        }
        want().safeTransfer(msg.sender, r);
    }
```

Proposed Recommendation: Implement a check to ensure that funds are available in the vault before proceeding with the withdrawal calculations. If necessary, adjust the logic to handle strategy withdrawals more efficiently.

ID	2
Title	Suboptimal Function Organization
Path	contracts/BayVault.sol
Function Name	*

Description: In the current contract, functions are not sorted in a manner that reflects their operational flow or categorization (like view functions, external functions, internal logic, etc.). This disorganization can lead to confusion about how the contract is supposed to operate, making it difficult for developers and auditors to trace the contract's logic and ensure its security and efficiency.

Impact: While this issue does not pose a direct security risk, it significantly impacts the maintainability and auditability of the contract. Poorly organized code can lead to misunderstandings, increasing the likelihood of bugs in future updates or modifications. It may also hinder the ability of new developers or auditors to understand and assess the contract's functionality quickly, thus indirectly raising the risk of overlooking potential vulnerabilities.

Code Reference: General observation throughout the contract.

Proposed Recommendation: Reorganize functions in a logical order, such as grouping view functions, external functions, and internal logic separately.

ID	3
Title	Non-Intuitive Variable Naming
Path	contracts/BayVault.sol
Function Name	withdraw( )

Description: The use of non-descriptive variable names like r, b, _diff, etc., in critical functions like withdraw makes the code less readable and harder to understand. This lack of clarity can lead to misinterpretation of the contract's logic, especially in complex functions where the context and role of each variable are crucial for understanding the flow and implications of the code.

Impact: This naming issue impacts the contract's maintainability and auditability increasing the risk of errors during future updates or maintenance, as developers may misinterpret the variables' purposes. This risk is especially high for resources who are not familiar with the original developer's naming conventions.

Code Reference:

```
function withdraw(uint256 _shares) public {
    uint256 r = (balance() * _shares) / totalSupply();
    _burn(msg.sender, _shares);

    uint b = want().balanceOf(address(this));
    if (b < r) {
        uint _withdraw = r - b;
        strategy.withdraw(_withdraw);
        uint _after = want().balanceOf(address(this));
        uint _diff = _after - b;
        if (_diff < _withdraw) {
            r = b + _diff;
        }
    }
    want().safeTransfer(msg.sender, r);
}
```

Proposed Recommendation: Use more descriptive variable names to enhance code clarity and maintainability.

ID	4
Title	Inefficient Use of External Function Visibility
Path	contracts/BayVault.sol
Function Name	proposeStrat( ), upgradeStrat( )

Description: The use of public visibility for functions like proposeStrat and upgradeStrat, which are not intended to be called internally, results in suboptimal gas usage. Marking these functions as external would be more appropriate, as external functions are optimized for gas when they are only called externally, owing to the way Solidity handles function arguments.

Impact: The direct impact of this inefficiency is increased transaction costs for these functions. Over time, these additional costs accumulate, especially in a contract where strategy updates might be frequent. This not only affects the economic efficiency of the contract but could also deter users or contract owners from executing these functions as often as might be needed, potentially affecting the contract's performance and adaptability.

Code Reference:

```
function proposeStrat(address _implementation) public onlyOwner { }
function upgradeStrat() public onlyOwner { }
```

Proposed Recommendation: Change the visibility of these functions to external to optimize gas usage.

DISCLAIMER

The smart contracts provided by the client with the purpose of security review have been thoroughly analyzed in compliance with the industrial best practices till date w.r.t. Smart Contract Weakness Classification (SWC) and Cybersecurity Vulnerabilities in smart contract code, the details of which are enclosed in this report.

This report is not an endorsement or indictment of the project or team, and they do not in any way guarantee the security of the particular object in context. This report is not considered, and should not be interpreted as an influence, on the potential economics of the token (if any), its sale, or any other aspect of the project that contributes to the protocol's public marketing.

Crypto assets/ tokens are the results of the emerging blockchain technology in the domain of decentralized finance and they carry with them high levels of technical risk and uncertainty. No report provides any warranty or representation to any third-party in any respect, including regarding the bug-free 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 the reports in any way, including to make any decisions to buy or sell any token, product, service, or 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 is not a guarantee as to the absolute security of the project.

Smart contracts are deployed and executed on a blockchain. The platform, its programming language, and other software related to the smart contract can have vulnerabilities that can lead to hacks. The scope of our review is limited to a review of the programmable code and only the programmable code, we note, as being within the scope of our review within this report. The smart contract programming 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 the programming language's compiler scope that could present security risks.

This security review cannot be considered a sufficient assessment regarding the utility and safety of the code, bug-free status, or any other statements of the contract. While BlockApex has done their best in conducting the analysis and producing this report, it is important to note that one should not rely on this report only - we recommend proceeding with several independent code security reviews and a public bug bounty program to ensure the security of smart contracts.