

Bitbond

Interim Smart Contract Audit Report

Contents

Revision History & Version Control

1.0 Disclaimer

2.0 Overview

- 2.1 Project Overview
- 2.2 Scope
- 2.3 Project Summary
- 2.4 Audit Summary
- 2.5 Security Level References
- 2.6 Vulnerability Summary

3.0 Executive Summary

- 3.1 Findings
- 3.2 Recommendations

4.0 Technical Analysis

- 4.1 Arbitrary "from" passed to transferFrom (or safeTransferFrom)
- 4.2 Centralization Risk for trusted owners
- 4.3 Functions not used internally could be marked external
- 4.4 No Zero-Value Check in increaseAllowance and decreaseAllowance Function
- 4.5 Prevention of Zero-Amount Check in Burn function in ERC20 Token Contract
- 4.6 Dead-code
- 4.7 State variables that could be declared immutable
- 4.8 Unnecessary Abstraction in ERC20 Token Implementation

5.0 Auditing Approach and Methodologies applied

- 5.1 Structural Analysis
- 5.2 Static Analysis
- 5.3 Code Review / Manual Analysis
- 5.4 Gas Consumption
- 5.5 Tools & Platforms Used For Audit
- 5.6 Checked Vulnerabilities

6.0 Limitations on Disclosure and Use of this Report

Revision History & Version Control

Start Date	End Date	Author	Comments/Details
25 Apr 2024	07 May 2024	Gurkirat	Interim Release for the Client

Reviewed by	Released by
Nelson Garnepudi	Nishita Palaksha

Entersoft was commissioned to perform a source code review on Bitbond's smart contracts. The review was conducted between April 25, 2024, to May 7, 2024. The report is organized into the following sections.

- Executive Summary: A high-level overview of the security audit findings.
- Technical analysis: Our detailed analysis of the Smart Contract code

The information in this report should be used to understand overall code quality, security, correctness, and meaning that code will work as described in the smart contract.

1.0 Disclaimer

This is a limited audit report on our findings based on our analysis, in accordance with good industry practice as at the date of this report, in relation to: (i) smart contract best coding practices and vulnerabilities in the framework and algorithms based on white paper, code, the details of which are set out in this report, (Smart Contract audit). To get a full view of our analysis, it is crucial for you to read the full report. While we have done our best in conducting our analysis and producing this report, it is important to note that you should not rely on this report and cannot claim against us based on what it says or does not say, or how we produced it, and it is important for you to conduct your own independent investigations before making any decisions. We go into more detail on this in the disclaimer below – please make sure to read it in full.

DISCLAIMER: By reading this report or any part of it, you agree to the terms of this disclaimer. If you do not agree to the terms, then please immediately cease reading this report, and delete and destroy any copies of this report downloaded and/or printed by you. This report is provided for information purposes only and on a non-reliance basis and does not constitute investment advice. No one shall have any right to rely on the report or its contents, and Entersoft Australia and its affiliates (including holding companies, shareholders, subsidiaries, employees, directors, officers, and other representatives) (Entersoft) owe no duty of care towards you or any other person, nor does Entersoft make any warranty or representation to any person on the accuracy or completeness of the report. The report is provided "as is", without any conditions, warranties or other terms of any kind except as set out in this disclaimer, and Entersoft hereby excludes all representations, warranties, conditions and other terms (including, without limitation, the warranties implied by law of satisfactory quality, fitness for purpose and the use of reasonable care and skill) which, but for this clause, might have effect in relation to the report. Except and only to the extent that it is prohibited by law, Entersoft hereby excludes all liability and responsibility, and neither you nor any other person shall have any claim against Entersoft, for any amount or kind of loss or damage that may result to you or any other person (including without limitation, any direct, indirect, special, punitive, consequential or pure economic loss or damages, or any loss of income, profits, goodwill, data, contracts, use of money, or business interruption, and whether in delict, tort (including without limitation negligence), contract, breach of statutory duty, misrepresentation (whether innocent or negligent) or otherwise under any claim of any nature whatsoever in any jurisdiction) in any way arising from or connected with this report and the use, inability to use or the results of use of this report, and any reliance on this report. The analysis of the Smart contract is purely based on the smart contract code shared with us alone.

2.0 Overview

2.1 Project Overview

During the period of **25 Apr 2024 to 07 May 2024**, Entersoft performed smart contract security audits for **Bitbond**.

2.2 Scope

The scope of this audit was to analyze and document the smart contract codebase for quality, security, and correctness.

The following files were reviewed as part of the scope:

- Context.sol
- LibCommon.sol
- Ownable.sol
- ERC20.sol
- ReflectiveERC20.sol
- DefiV3Token.sol

OUT-OF-SCOPE: External contracts, External Oracles, other smart contracts in the repository, or imported smart contracts.

1.

2.3 Project Summary

Project Name	No. of Smart Contract File(s)	Verified	Vulnerabilities
Bitbond	1	Yes	As per report. Section 2.6

2.4 Audit Summary

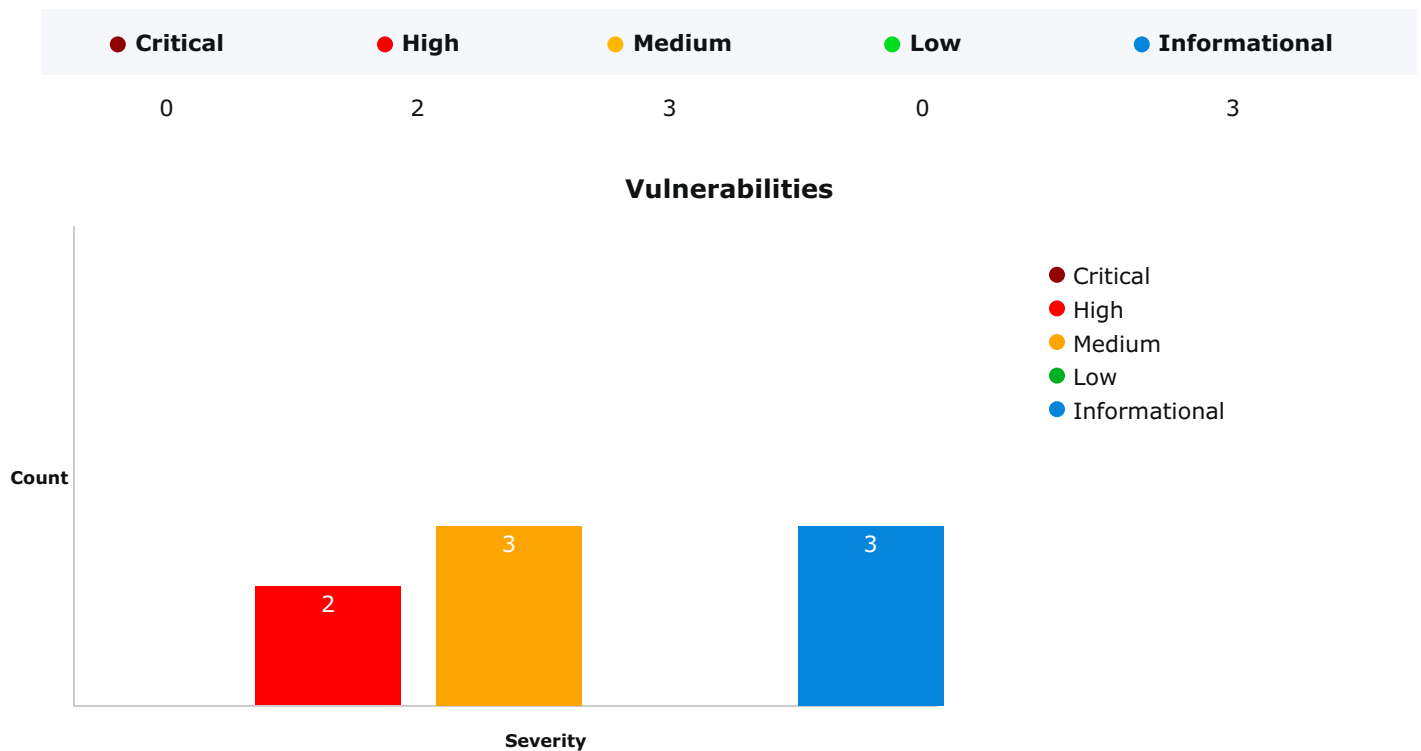
Delivery Date	Method of Audit	Consultants Engaged
07 May 2024	Manual and Automated approach	3

2.5 Security Level References

Every vulnerability in this report was assigned a severity level from the following classification table:

		Impact				
		Minimal	Low	Medium	High	Critical
Likelihood	Critical	Minimal	Low	Medium	High	Critical
	High	Minimal	Low	Medium	High	Critical
	Medium	Minimal	Low	Medium	Medium	High
	Low	Minimal	Low	Low	Low	Medium
	Minimal	Minimal	Minimal	Minimal	Low	Low

2.6 Vulnerability Summary



3.0 Executive Summary

Entersoft has conducted a comprehensive technical audit of the Bitbond smart contract through a comprehensive smart contract audit approach. The primary objective was to identify potential vulnerabilities and security risks within the codebase, ensuring adherence to industry-leading standards while prioritizing security, reliability, and performance. Our focus was on prompt and efficient identification and resolution of vulnerabilities to enhance the overall robustness of the solidity smart contract.

Importantly, our audit process intentionally avoided reliance solely on automated tools, emphasizing a more in-depth and nuanced approach to security analysis. Conducted from April 25, 2024, to May 7, 2024, our team diligently assessed and validated the security posture of the solidity smart contract, ultimately finding a number of vulnerabilities as per vulnerability summary table.

Testing Methodology:

We have leveraged static analysis techniques extensively to identify potential vulnerabilities automatically with the aid of cutting-edge tools such as Slither and Aderyn. Apart from this, we carried out extensive manual testing to iron out vulnerabilities that could slip through an automated check. This included a variety of attack vectors like reentrancy attacks, overflow and underflow attacks, timestamp dependency attacks, and more.

While going through the due course of this audit, we also ensured to cover edge cases, and built a combination of scenarios to assess the contracts' resilience. Our attempt to leave no stone unturned involved coming up with both negative and positive test cases for the system, and grace handling of stressed scenarios.

Our testing methodology in Solidity adhered to industry standards and best practices, integrating partially implemented OWASP and NIST SP 800 standards for encryption and signatures. Solidity's renowned security practices were complemented by tools such as Solhint for linting, and the Solidity compiler for code optimization. Sol-profiler, Sol-coverage, and Sol-sec were employed to ensure code readability and eliminate unnecessary dependencies.

Tools Used for Audit:

In the course of our audit, we leveraged a suite of tools to bolster the security and performance of our program. While our team drew on their expertise and industry best practices, we also integrated various tools into our development environment. Noteworthy among them are Slither, aderyn. This holistic approach ensures a thorough analysis, uncovering potential issues that automated tools alone might overlook. Entersoft takes pride in utilizing these tools, which significantly contribute to the quality, security, and maintainability of our codebase.

Code Review / Manual Analysis:

Our team conducted a manual analysis of the Solidity smart contracts to identify new vulnerabilities or to verify vulnerabilities found during static and manual analysis. We carefully analyzed every line of code and made sure that all instructions provided during the onboarding phase were followed. Through our manual analysis, we were able to identify potential vulnerabilities that may have been missed by automated tools and

ensure that the smart contract was secure and reliable.

Auditing Approach and Methodologies Applied:

The solidity smart contract was audited in a comprehensive approach to ensure the highest level of security and reliability. Careful attention was given to the following key areas to ensure the overall quality of code:

- **Code quality and structure:** We conducted a detailed review of the codebase to identify any potential issues related to code structure, readability, and maintainability. This included analyzing the overall architecture of the solidity smart contract and reviewing the code to ensure it follows best practices and coding standards.
- **Security vulnerabilities:** Our team used manual techniques to identify any potential security vulnerabilities that could be exploited by attackers. This involved a thorough analysis of the code to identify any potential weaknesses, such as buffer overflows, injection vulnerabilities, Signatures, and deprecated functions.

3.1 Findings

Vulnerability ID	Contract Name	Severity	Status
1	DefiV3Token.sol	● High	Pending
2	DefiV3Token.sol, Ownable.sol	● High	Pending
3	DefiV3Token.sol, ERC20.sol, Ownable.sol, ReflectiveERC20.sol	● Medium	Pending
4	ERC20.sol	● Medium	Pending
5	ERC20.sol	● Medium	Pending
6	LibCommon.sol	● Informational	Pending
7	DefiV3Token.sol	● Informational	Pending
8	ERC20.sol, ReflectiveERC20.sol	● Informational	Pending

3.2 Recommendations

Overall, the smart contracts are very well written, and they adhere to best security practices and industry

guidelines.

4.0 Technical Analysis

4.1 Arbitrary "from" passed to transferFrom (or safeTransferFrom)

Severity	Status	Type of Analysis
● High	Identified	Static

Contract Name:

DefiV3Token.sol

Description:

Passing an arbitrary `from` address to `transferFrom` (or `safeTransferFrom`) can lead to loss of funds, because anyone can transfer tokens from the `from` address if an approval is made.

Locations:

Line number: 366

Remediation:

To address the vulnerability, validate the `from` address before executing `transferFrom`. Implement access control mechanisms like role-based access or permissioned token standards such as ERC-777. Additionally, consider security measures like rate limiting or extra authorization steps for sensitive transfers to mitigate risks.

Impact:

If this vulnerability is not fixed, it could result in illegal token transfers, which would hurt the token's and its ecosystem's reputation in addition to costing token holders money. To maintain protocol integrity and safeguard user assets, token transfer security must be given top priority.

Code Snippet:

```
function transferFrom(
    address from,
    address to,
    uint256 amount
) public virtual override returns (bool) {
    uint256 taxAmount = _taxAmount(from, amount);
    uint256 deflationAmount = _deflationAmount(amount);
    uint256 amountToTransfer = amount - taxAmount - deflationAmount;
    if (isMaxAmountOfTokensSet()) {
        if (balanceOf(to) + amountToTransfer > maxTokenAmountPerAddress) {
            revert DestBalanceExceedsMaxAllowed(to);
        }
    }
    if (taxAmount != 0) {
        _transferNonReflectedTax(from, taxAddress, taxAmount);
    }
    if (deflationAmount != 0) {
        _burn(from, deflationAmount);
    }
    return super.transferFrom(from, to, amountToTransfer);
}
```

Reference:**Proof of Vulnerability:**

N.A.

4.2 Centralization Risk for trusted owners

Severity	Status	Type of Analysis
● High	Identified	Static

Contract Name:

DefiV3Token.sol, Ownable.sol

Description:

Contracts have owners with privileged rights to perform admin tasks and need to be trusted to not perform malicious updates or drain funds.

Locations:

Line number:

- DefiV3Token.sol: 11, 236, 249, 264, 280, 298, 373, 394, 403, 410.
- Ownable.sol: 61, 69.

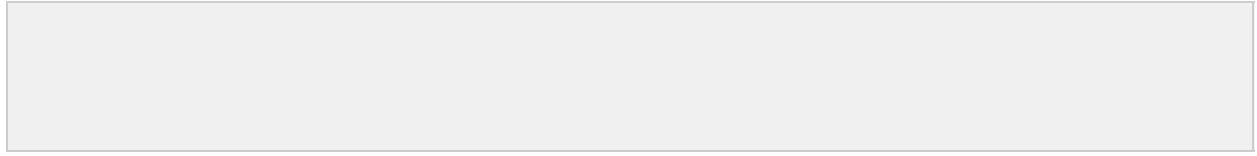
Remediation:

Examine and modify functions that use the onlyOwner modification to reduce the powers granted to owners in order to mitigate the risk of centralization. In order to decentralise decision-making, implement multi-signature procedures for important administrative activities and investigate community governance models. The protocol can reduce the risks associated with centralization and improve resistance against possible hostile activities by dispersing control and implementing safeguards.

Impact:

The decentralisation ideals of the protocol may be compromised by the centralization threats posed by the existence of trusted owners with broad powers. A compromised or malicious owner's malicious activities could result in money loss or detrimental protocol changes, undermining trust and discouraging participation in the ecosystem.

Code Snippet:

**Reference:**

<https://www.certik.com/resources/blog/What-is-centralization-risk>

Proof of Vulnerability:

N.A.

4.3 Functions not used internally could be marked external

Severity	Status	Type of Analysis
● Medium	Identified	Static

Contract Name:

DefiV3Token.sol, ERC20.sol, Ownable.sol, ReflectiveERC20.sol

Description:

Functions that are not called internally within the contract and are intended to be called externally by users or other contracts should be marked as external instead of public for clarity and potentially a slight optimization.

Locations:

Line number:

- DefiV3Token.sol: 211, 315, 344, 403.
- ERC20.sol: 62, 70, 87, 101, 113, 136,158, 177,197.
- Ownable.sol: 61, 69.
- ReflectiveERC20.sol: 86, 101

Remediation:

Change the visibility of these functions to external if they are intended to be called from outside the contract. This improves readability and provides a hint to developers about the intended usage.

Impact:

The mentioned functions are declared as public but are not called internally within the contract. They are likely meant to be called externally.

Code Snippet:

```
//DefiV3Token.sol: 211

function decimals() public view virtual override returns (uint8) {
    return _decimals;
}
```

```
//DefiV3Token.sol: 315

function transfer(
    address to,
    uint256 amount
) public virtual override returns (bool) {
    uint256 taxAmount = _taxAmount(msg.sender, amount);
    uint256 deflationAmount = _deflationAmount(amount);
    uint256 amountToTransfer = amount - taxAmount - deflationAmount;
    if (isMaxAmountOfTokensSet()) {
        if (balanceOf(to) + amountToTransfer > maxTokenAmountPerAddress) {
            revert DestBalanceExceedsMaxAllowed(to);
        }
    }
    if (taxAmount != 0) {
        _transferNonReflectedTax(msg.sender, taxAddress, taxAmount);
    }
    if (deflationAmount != 0) {
        _burn(msg.sender, deflationAmount);
    }
    return super.transfer(to, amountToTransfer);
}

//DefiV3Token.sol: 344

function transferFrom(
    address from,
    address to,
    uint256 amount
) public virtual override returns (bool) {
    uint256 taxAmount = _taxAmount(from, amount);
    uint256 deflationAmount = _deflationAmount(amount);
```

```

uint256 amountToTransfer = amount - taxAmount - deflationAmount;

if (isMaxAmountOfTokensSet()) {

if (balanceOf(to) + amountToTransfer > maxTokenAmountPerAddress) {

revert DestBalanceExceedsMaxAllowed(to);

}

}

if (taxAmount != 0) {

_transferNonReflectedTax(from, taxAddress, taxAmount);

}

if (deflationAmount != 0) {

_burn(from, deflationAmount);

}

return super.transferFrom(from, to, amountToTransfer);

}

//DefiV3Token.sol: 403

function renounceOwnership() public override onlyOwner {

super.renounceOwnership();

}

//ERC20.sol: 62

function name() public view virtual override returns (string memory) {

return _name;

}

//ERC20.sol: 70

function symbol() public view virtual override returns (string memory) {

return _symbol;

}

```



```
//ERC20.sol: 87

function decimals() public view virtual override returns (uint8) {

return 18;

}


//ERC20.sol: 101

function balanceOf(address account) public view virtual override returns (uint256) {

return _balances[account];

}


//ERC20.sol: 113

function transfer(address to, uint256 amount) public virtual override returns (bool) {

address owner = _msgSender();

_transfer(owner, to, amount);

return true;

}


//ERC20.sol: 136

function approve(address spender, uint256 amount) public virtual override returns (bool) {

address owner = _msgSender();

_approve(owner, spender, amount);

return true;

}


//ERC20.sol: 158

function transferFrom(address from, address to, uint256 amount) public virtual override returns (bool) {

address spender = _msgSender();

_spendAllowance(from, spender, amount);

_transfer(from, to, amount);
```

```
return true;

}

//ERC20.sol: 177

function increaseAllowance(address spender, uint256 addedValue) public virtual returns (bool) {
    address owner = _msgSender();
    _approve(owner, spender, allowance(owner, spender) + addedValue);
    return true;
}

//ERC20.sol: 197

function decreaseAllowance(address spender, uint256 subtractedValue) public virtual returns (bool) {
    address owner = _msgSender();

    uint256 currentAllowance = allowance(owner, spender);
    require(currentAllowance >= subtractedValue, "ERC20: decreased allowance below zero");

    unchecked {
        _approve(owner, spender, currentAllowance - subtractedValue);
    }

    return true;
}

//Ownable.sol: 61

function renounceOwnership() public virtual onlyOwner {
    _transferOwnership(address(0));
}

//Ownable.sol: 69

function transferOwnership(address newOwner) public virtual onlyOwner {
    require(newOwner != address(0), "Ownable: new owner is the zero address");
    _transferOwnership(newOwner);
}
```

```

}

//ReflectiveERC20.sol: 86

function transferFrom(
    address from,
    address to,
    uint256 value
) public virtual override returns (bool) {
    address spender = super._msgSender();
    _spendAllowance(from, spender, value);
    _transfer(from, to, value);
    return true;
}

//ReflectiveERC20.sol: 101

function transfer(
    address to,
    uint256 value
) public virtual override returns (bool) {
    address owner = super._msgSender();
    _transfer(owner, to, value);
    return true;
}

```

Reference:

<https://github.com/crytic/slither/wiki/Detector-Documentation#public-function-that-could-be-declared-external>

Proof of Vulnerability:

N.A.

4.4 No Zero-Value Check in increaseAllowance and decreaseAllowance Function

Severity	Status	Type of Analysis
● Medium	Identified	Dynamic

Contract Name:

ERC20.sol

Description:

The increaseAllowance function allows for the addition of a zero addedValue to the existing allowance, which results in an unnecessary approval transaction. This consumes gas without any change to the allowance state, potentially leading to inefficient gas usage.

Locations:

ERC20.sol, Line: 177-181, 197-206

Remediation:

Introduce a requirement to check that addedValue is greater than zero in the increaseAllowance function and decreaseAllowance. This can be achieved by adding a condition such as require(addedValue > 0, "ERC20: addedValue must be greater than zero");

Impact:

Calling increaseAllowance with a zero addedValue does not alter the state but still causes the execution of state-modifying operations (e.g., emitting an Approval event), which leads to unnecessary gas expenditure.

Code Snippet:

```
//Function 1 Line: 177-181

function increaseAllowance(address spender, uint256 addedValue) public virtual returns (bool) {
    address owner = _msgSender();
    _approve(owner, spender, allowance(owner, spender) + addedValue);
    return true;
}

//Function 1 Line: 197-206

function decreaseAllowance(address spender, uint256 subtractedValue) public virtual returns (bool) {
    address owner = _msgSender();
    uint256 currentAllowance = allowance(owner, spender);
    require(currentAllowance >= subtractedValue, "ERC20: decreased allowance below zero");
    unchecked {
        _approve(owner, spender, currentAllowance - subtractedValue);
    }
    return true;
}
```

Reference:**Proof of Vulnerability:**

N.A.

4.5 Prevention of Zero-Amount Check in Burn function in ERC20 Token Contract

Severity	Status	Type of Analysis
● Medium	Identified	Dynamic

Contract Name:

ERC20.sol

Description:

The **_burn** function allows for burning zero tokens, which results in gas expenditure without any change in token state or total supply. While this does not pose a security risk, it could lead to unnecessary gas consumption.

Locations:

ERC20.sol line 277 to 293

Remediation:

Consider adding a **require(amount != 0, "ERC20: Cannot burn zero amount");** at the beginning of the **_burn** function to prevent burning of zero tokens and to ensure that every burn operation leads to a meaningful state change. This practice can enhance contract efficiency and prevent users from executing pointless transactions.

This added check enhances the economic efficiency of the contract by avoiding pointless gas expenditure on no-op transactions.

Impact:

NA


Code Snippet:

```
function _burn(address account, uint256 amount) internal virtual {  
  
    require(account != address(0), "ERC20: burn from the zero address");  
  
    _beforeTokenTransfer(account, address(0), amount);  
  
    uint256 accountBalance = _balances[account];  
  
    require(accountBalance >= amount, "ERC20: burn amount exceeds balance");  
  
    unchecked {  
  
        _balances[account] = accountBalance - amount;  
  
        // Overflow not possible: amount <= accountBalance <= totalSupply.  
  
        _totalSupply -= amount;  
  
    }  
  
    emit Transfer(account, address(0), amount);  
  
    _afterTokenTransfer(account, address(0), amount);  
  
}
```

Reference:**Proof of Vulnerability:**

N.A.

4.6 Dead-code

Severity	Status	Type of Analysis
 Informational	Identified	Static

Contract Name:

LibCommon.sol

Description:

Functions that are not sued.

Locations:

Line number:

- LibCommon.sol: 95-117.
- LibCommon.sol: 25-37.
- LibCommon.sol: 60-88.

Remediation:

Remove unused functions.

Impact:

Having dead code, like unused functions, in a contract complicates code review and maintenance without providing any functionality.

Code Snippet:

```
//LibCommon.sol: 95-117

function safeTransfer(
    address tokenAddress,
    address to,
    uint256 amount
) internal returns (bool) {
    // solhint-disable-next-line avoid-low-level-calls
    (bool success, bytes memory data) = tokenAddress.call(
```

```

(0001 success, bytes memory data) = tokenAddress.call(

abi.encodeWithSignature("transfer(address,uint256)", to, amount)

);

if (!success) {

if (data.length != 0) {

// bubble up error

// solhint-disable-next-line no-inline-assembly

assembly {

let returndata_size := mload(data)

revert(add(32, data), returndata_size)

}

} else {

revert TransferFailed();

}

}

return true;

}

//LibCommon.sol: 25-37

function safeTransferETH(address to, uint256 amount) internal {

// solhint-disable-next-line no-inline-assembly

assembly {

// Transfer the ETH and check if it succeeded or not.

if iszero(call(gas(), to, amount, 0, 0, 0, 0)) {

// Store the function selector of `ETHTransferFailed()`.

// bytes4(keccak256(bytes("ETHTransferFailed()"))) = 0xb12d13eb

mstore(0x00, 0xb12d13eb)

// Revert with (offset, size).

revert(0x1c, 0x04)

}

```

```

}

}

//LibCommon.sol: 60-88.

function safeTransferFrom(
    address tokenAddress,
    address from,
    address to,
    uint256 amount
) internal returns (bool) {
    (bool success, bytes memory data) = tokenAddress.call(
        abi.encodeWithSignature(
            "transferFrom(address,address,uint256)",
            from,
            to,
            amount
        )
    );
    if (!success) {
        if (data.length != 0) {
            // bubble up error
            // solhint-disable-next-line no-inline-assembly
            assembly {
                let returndata_size := mload(data)
                revert(add(32, data), returndata_size)
            }
        } else {
            revert TransferFailed();
        }
    }
}

```

```
return true;  
  
}
```


Reference:

<https://github.com/crytic/slither/wiki/Detector-Documentation#dead-code>

Proof of Vulnerability:

N.A.

4.7 State variables that could be declared immutable

Severity	Status	Type of Analysis
 Informational	Identified	Static

Contract Name:

DefiV3Token.sol

Description:

State variables that are not updated following deployment should be declared immutable to save gas.

Locations:

Line number: 23

Remediation:

Add the immutable attribute to state variables that never change or are set only in the constructor.

Impact:

Gas costs can be inadvertently increased by not designating state variables as immutable when they remain unchanged after deployment. Designating certain variables as immutable increases contract performance and scalability, and decreases gas consumption during deployment and interactions.

Code Snippet:

```
uint256 public maxTotalSupply;
```

Reference:

<https://github.com/crytic/slither/wiki/Detector-Documentation#state-variables-that-could-be-declared-immutable>

Proof of Vulnerability:

N.A.

4.8 Unnecessary Abstraction in ERC20 Token Implementation

Severity	Status	Type of Analysis
● Informational	Identified	Dynamic

Contract Name:

ERC20.sol, ReflectiveERC20.sol

Description:

The ERC20 token contract uses `_msgSender()` to abstract the `msg.sender` for potential meta-transaction handling. In scenarios where meta-transactions are not utilized, this abstraction might be considered unnecessary, slightly increasing complexity and gas costs per transaction.

Locations:

ERC20.sol, Line no - 113-117, 136-140, 158-163, 177-181, 197-206

ReflectiveERC20.sol, Line no - 86-95, 101-108

Remediation:

For contracts where meta-transactions are not required, consider using `msg.sender` directly to simplify the contract and potentially save gas. This is particularly applicable in private or permissioned blockchain scenarios where transaction initiators are always the message senders.

Impact:

The direct use of **`msg.sender`** instead of **`_msgSender()`** might marginally decrease gas costs and simplify the code where meta-transactions are not a concern. However, it reduces the contract's flexibility to be adapted to future use-cases involving meta-transactions.

Code Snippet:

```
//ERC20.sol

//Function 1, Line No -- 113-117
```

```
function transfer(address to, uint256 amount) public virtual override returns (bool) {  
    address owner = _msgSender();  
    _transfer(owner, to, amount);  
    return true;  
}  
  
//Function 2, Line No -- 136-140  
  
function approve(address spender, uint256 amount) public virtual override returns (bool) {  
    address owner = _msgSender();  
    _approve(owner, spender, amount);  
    return true;  
}  
  
//Function 3, Line No -- 158-163  
  
function transferFrom(address from, address to, uint256 amount) public virtual override returns (bool) {  
    address spender = _msgSender();  
    _spendAllowance(from, spender, amount);  
    _transfer(from, to, amount);  
    return true;  
}  
  
Function 4, Line No -- 177-181  
  
function increaseAllowance(address spender, uint256 addedValue) public virtual returns (bool) {  
    address owner = _msgSender();  
    approve(owner, spender, allowance(owner, spender) + addedValue);  
}
```

```

return true;
}

//Function 5, Line No -- 197-206

function decreaseAllowance(address spender, uint256 subtractedValue) public virtual returns (bool) {
    address owner = _msgSender();
    uint256 currentAllowance = allowance(owner, spender);
    require(currentAllowance >= subtractedValue, "ERC20: decreased allowance below zero");
    unchecked {
        _approve(owner, spender, currentAllowance - subtractedValue);
    }
    return true;
}

//ReflectiveERC20.sol

//Function 1, Line No -- 86-95

function transferFrom(
    address from,
    address to,
    uint256 value
) public virtual override returns (bool) {
    address spender = super._msgSender();
    _spendAllowance(from, spender, value);
    _transfer(from, to, value);
    return true;
}

```



```
}

//Funciton 2, Line No -- 101-108

function transfer(
address to,
uint256 value
) public virtual override returns (bool) {
address owner = super._msgSender();
_transfer(owner, to, value);
return true;
}
```

Reference:**Proof of Vulnerability:**

N.A.

5.0 Auditing Approach and Methodologies applied

Throughout the audit of the smart contract, care was taken to ensure:

- Overall quality of code
- Use of best practices.
- Code documentation and comments match logic and expected behavior.
- Mathematical calculations are as per the intended behavior mentioned in the whitepaper.
- Implementation of token standards.
- Efficient use of gas.
- Code is safe from Re-entrancy and other vulnerabilities.

A combination of manual and automated security testing to balance efficiency, timeliness, practicality, and accuracy regarding the scope of the smart contract audit. While manual testing is recommended to uncover flaws in logic, process, and implementation; automated testing techniques help enhance coverage of smart contracts and can quickly identify items that do not follow security best practices. The following phases and associated tools were used throughout the term of the audit:

5.1 Structural Analysis

In this step we have analysed the design patterns and structure of all smart contracts. A thorough check was completed to ensure all Smart contracts are structured in a way that will not result in future problems.

5.2 Static Analysis

Static Analysis of smart contracts was undertaken to identify contract vulnerabilities. In this step, a series of automated tools are used to test the security of smart contracts.

5.3 Code Review / Manual Analysis

Manual Analysis or review of done to identify new vulnerabilities or to verify the vulnerabilities found during the Static Analysis. The contracts were completely manually analysed, and their logic was checked and compared with the one described in the whitepaper. It should also be noted that the results of the automated analysis were verified manually.

5.4 Gas Consumption

In this step, we checked the behaviour of all smart contracts in production. Checks were completed to understand how much gas gets consumed, along with the possibilities of optimisation of code to reduce gas consumption.

5.5 Tools & Platforms Used For Audit

Slither, Aderyn

5.6 Checked Vulnerabilities

We have scanned Bitbond smart contracts for commonly known and more specific vulnerabilities. Here are some of the commonly known vulnerabilities that we considered:

- Re-entrancy
- Timestamp Dependence
- Gas Limit and Loops
- DoS with Block Gas Limit
- Transaction-Ordering Dependence
- Use of tx.origin
- Exception disorder
- Gasless send
- Balance equality
- Byte array
- Transfer forwards all gas
- ERC-20 API violation
- Malicious libraries
- Compiler version not fixed
- Redundant fallback function
- Send instead of transfer
- Style guide violation
- Unchecked external call
- Unchecked math
- Unsafe type inference
- Implicit visibility level

6.0 Limitations on Disclosure and Use of this Report

This report contains information concerning potential details of Bitbond and methods for exploiting them. Entersoft recommends that special precautions be taken to protect the confidentiality of both this document and the information contained herein. Security Assessment is an uncertain process, based on past experiences, currently available information, and known threats. All information security systems, which by their nature are dependent on human beings, are vulnerable to some degree. Therefore, while Entersoft considers the major security vulnerabilities of the analyzed systems to have been identified, there can be no assurance that any exercise of this nature will identify all possible vulnerabilities or propose exhaustive and operationally viable recommendations to mitigate those exposures. In addition, the analysis set forth herein is based on the technologies and known threats as of the date of this report. As technologies and risks change over time, the vulnerabilities associated with the operation of the Smart Contract described in this report, as well as the actions necessary to reduce the exposure to such vulnerabilities will also change. Entersoft makes no undertaking to supplement or update this report based on changed circumstances or facts of which Entersoft becomes aware after the date hereof, absent a specific written agreement to perform the supplemental or updated analysis. This report may recommend that Entersoft use certain software or hardware products manufactured or maintained by other vendors. Entersoft bases these recommendations upon its prior experience with the capabilities of those products. Nonetheless, Entersoft does not and cannot warrant that a particular product will work as advertised by the vendor, nor that it will operate in the manner intended. This report was prepared by Entersoft for the exclusive benefit of Bitbond and is proprietary information. The Non-Disclosure Agreement (NDA) in effect between Entersoft and Bitbond governs the disclosure of this report to all other parties including product vendors and suppliers.