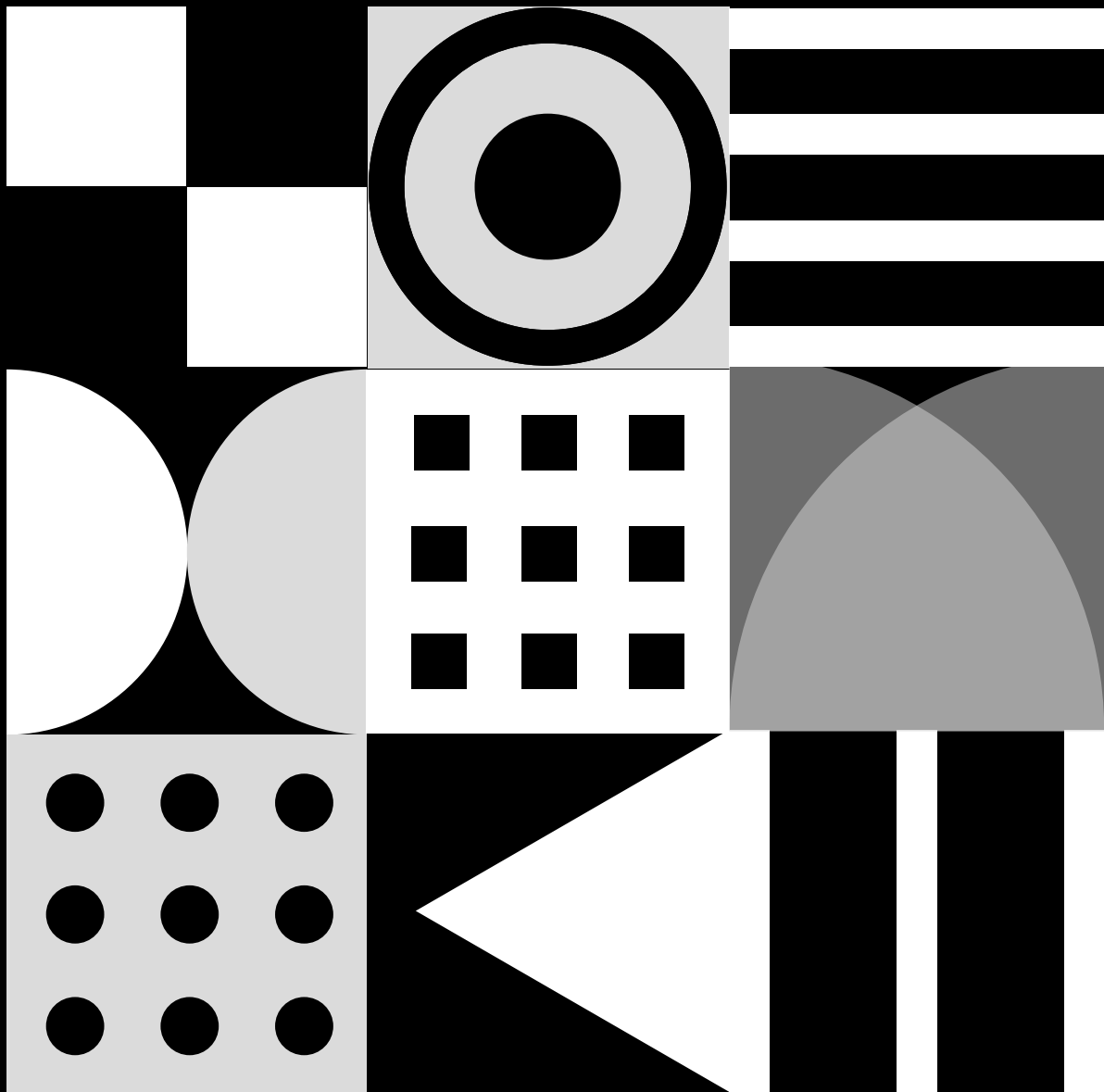



INSIDIOUS AUDIT REPORT

BAFI FINANCE

18908



Contents

- 1. Introduction
 - 1.1. About Project
 - 1.2. Audit Goal
 - 1.3. Disclaimer
- 2. Findings
 - 2.1. Data Validation Issues - **PASS**
 - 2.2. Random Number Issues - **PASS**
 - 2.3. State Issues - **PASS**
 - 2.4. Error Conditions, Return Values, Status Codes - **PASS**
 - 2.5. Data Processing Errors - **PASS**
 - 2.6. Bad Coding Practices - **PASS**
 - 2.7. Permission Issues - **PASS**
 - 2.8. Business Logic Errors - **PASS**
 - 2.9. Complexity Issues - **PASS**
- 3. Conclusion
- 4. Appendix
 - 4.1. Proper Implementation 

1. Introduction

1.1. About Project

- Project Name: **BAFI TOKEN**
- We audited **BAFI** smart contract deployed at <https://bscscan.com/address/0xa2f46fe221f34dac4cf078e6946a7cb4e373ad28>. The **BAFI** is a standard
- BEP20 token without any business logic.
- The total supply is 100,000 and no emission or mint functions exist after initialization.

1.2. Audit Goal

Category	Content	Result
Data Validation Issues	Incorrect Behavior Order: Early Validation, Permissive List of AllowedInputs, Unchecked Input for Loop Condition	PASS
Random Number Issues	Small Space of Random Values, Incorrect Usage of Seeds in Pseudo-RandomNumber Generator (PRNG)	PASS
State Issues	External Control of System or Configuration Setting, Incomplete Internal StateDistinction, Passing Mutable Objects to an Untrusted Method	PASS
Error Conditions, Return Values, Status Codes	Unchecked Return Value, Unexpected Status Code or Return Value, ReachableAssertion, Detection of Error Condition Without Action	PASS
Data Processing Errors	Collapse of Data into Unsafe Value, Improper Handling of Parameters, Comparison of Incompatible Types	PASS
Bad Coding Practices	Missing Default Case in Switch Statement, Excessive Index Range Scan for aData Resource, Excessive Platform Resource Consumption within a Loop	PASS
Permission Issues	Incorrect Default Permissions, Incorrect Execution-AssignedPermissions, Improper Preservation of Permissions	PASS
Business Logic Errors	Unverified Ownership, Incorrect Ownership Assignment, Allocation ofResources Without Limits or Throttling	PASS
Complexity Issues	Loop Condition Value Update within the Loop, Excessively Deep Nesting	PASS

1.3. Disclaimer

Note that this audit does not promise that all possible security concerns with the specified smart contract(s) will be discovered; in other words, the evaluation result does not guarantee that no more security vulnerabilities will be discovered. Because a single audit-based assessment cannot be considered thorough, we always propose conducting multiple independent audits and participating in a public bug bounty program to assure smart contract security (s). Finally, this security assessment should not be construed as investment advice.

2. Findings

2.1. Data Validation Issues - **PASS**

This category includes flaws in a software system's input validation, output validation, and other types of validation components. Validation is a common approach for checking that data meets expectations before being processed further as input or output. Validation comes in a variety of forms. Although developers may view all attempts to provide "safe" inputs or outputs as validation, validation is separate from other strategies that attempt to modify data before processing it. Regardless, validation is a powerful technique that is frequently used to prevent malformed input from entering the system, as well as to indirectly eliminate code injection and other potentially harmful patterns while creating output. If not corrected, the flaws in this category may result in a decrease in the quality of data flow in a system.

Test results: vulnerabilities not detected in smart contract code.

Recommendation: None.

2.2. Random Number Issues - **PASS**

This category contains flaws in a software system's random number generation. Because the amount of possible random values is less than what the product requires, it is more vulnerable to brute force attacks. The code employs a Pseudo-Random Number Generator (PRNG) that fails to manage seeds properly.

Test results: vulnerabilities not detected in smart contract code.

Recommendation: None.

2.3. State Issues - **PASS**

This group of flaws is characterized by poor system state management. A user can control one or more system settings or configuration items from the outside. The software does not correctly determine which state it is in, leading it to believe it is in state X when it is in state Y, causing it to perform security-relevant activities incorrectly.

Test results: vulnerabilities not detected in smart contract code.

Recommendation: None.

2.4. Error Conditions, Return Values, Status Codes - **PASS**

This category contains flaws that arise when a function does not generate the correct return/status code, or when the application does not handle all of the various return/status codes that a function could generate. This type of issue is most identified in conditions that

are only experienced infrequently during the product's usual functioning. Most issues linked to common conditions are probably discovered and fixed during development and testing. In some circumstances, the attacker has direct control over or influence over the environment, causing the uncommon conditions to occur.

Test results: vulnerabilities not detected in smart contract code.

Recommendation: None.

2.5. Data Processing Errors - **PASS**

Weaknesses in this area are usually identified in data processing functionality. The manipulation of input to obtain or save information is known as data processing. The software filters data in such a way that it is reduced or "collapsed" into an unsafe value that violates a security property that is intended. When the expected number of values for parameters, fields, or arguments is not provided in input, or if those values are undefined, the software fail.

Test results: vulnerabilities not detected in smart contract code.

Recommendation: None.

2.6. Bad Coding Practices - **PASS**

Weaknesses in this category are associated with hazardous coding techniques that raise the likelihood of an exploitable vulnerability being present in the program. These flaws do not directly pose a vulnerability, but they do show that the product was not produced or maintained with care. If a program is complicated, difficult to maintain, not portable, or displays signs of neglect, there is a greater chance that flaws are hidden in the code.

Test results: vulnerabilities not detected in smart contract code.

Recommendation: None.

2.7. Permission Issues - **PASS**

Permissions are incorrectly assigned or handled, which is a weakness in this area. The software sets the permissions of an object in a way that is incompatible with the user's intended permissions while it is running. When copying, restoring, or sharing objects, the software does not retain permissions or does so erroneously, resulting in less restrictive permissions than intended.

Test results: vulnerabilities not detected in smart contract code.

Recommendation: None.

2.8. Business Logic Errors - **PASS**

This category of flaws identifies some of the underlying issues that allow attackers to modify an application's business logic. Business logic errors can be disastrous for an entire application. Because they usually entail acceptable use of the application's capabilities, they can be difficult to detect automatically. Many business logic problems, on the other hand, can show patterns that are comparable to well-known implementation and design flaws.

Test results: vulnerabilities not detected in smart contract code.

Recommendation: None.

2.9. Complexity Issues - **PASS**

Overcomplicated objects are related with this group of flaws. The code contains a callable or other code grouping in which the nesting / branching is too deep, or it employs a loop with a control flow condition dependent on a value that is modified within the body of the loop.

Test results: vulnerabilities not detected in smart contract code.

Recommendation: None.

3. Conclusion

Use case of the smart contract is simple and the code is relatively small. No security issues from external attackers were identified and therefore the contract is good to be deployed on public networks as per the audit team's analysis.

4. Appendix

4.1. Proper Implementation

```

library SafeMath {
    /**
     * @dev Returns the addition of two unsigned integers, reverting on
     * overflow.
     *
     * Counterpart to Solidity's '+' operator.
     *
     * Requirements:
     * - Addition cannot overflow.
     */
    function add(uint256 a, uint256 b) internal pure returns (uint256) {
        uint256 c = a + b;
        require(c >= a, "SafeMath: addition overflow");

        return c;
    }
}

```

This protects you from underflow and overflow attacks.

```

/**
 * @dev Transfers ownership of the contract to a new account (`newOwner`).
 */
function _transferOwnership(address newOwner) internal {
    require(newOwner != address(0), "Ownable: new owner is the zero address");
    emit OwnershipTransferred(_owner, newOwner);
    _owner = newOwner;
}
}

```

newOwner address value is properly checked.

```

function _burn(address account, uint256 amount) internal {
    require(account != address(0), "BEP20: burn from the zero address");

    _balances[account] = _balances[account].sub(amount, "BEP20: burn amount exceeds balance");
    _totalSupply = _totalSupply.sub(amount);
    emit Transfer(account, address(0), amount);
}

```

_burn function here we are checking that token can only be burnt and not new emission

Compiler version is not latest

=> In this file you have put “pragma solidity ^0.6.12;” which is not a good way to define compiler version.

=> Solidity source files indicate the versions of the compiler they can be compiled with.

```
Pragma solidity ^0.5.16;
```

```
// bad: compiles ^0.5.16 and above pragma solidity 0.5.16; //good: compiles 0.5.16 only
```

=> If you put(^) symbol then you are able to get compiler version 0.5.16 and above. But if you don't use(^) symbol then you are able to use only 0.5.16 version. And if there are some changes come in the compiler and you use the old version then some issues may come at deploy time.

=> Solidity latest version is 0.8.X

Summary of the Audit

Overall, the code is well and performs well. Please try to check the address and value of token externally before sending to the solidity code. Our final recommendation would be to pay more attention to the visibility of the functions, hardcoded address, and mapping since it's quite important to define who's supposed to execute the functions and to follow best practices regarding the use of assert, require etc. (which you are doing ;)).

Note: Please focus on version of solidity (Use latest) and check addresses