Ether Authority

www.EtherAuthority.io
audit@etherauthority.io

SMART CONTRACT

Security Audit Report

Project:	AI Done For You
Website:	aidfy.io
Platform:	Ethereum
Language:	Solidity
Date:	May 28th, 2025

Table of contents

Email: audit@EtherAuthority.io

`

THIS IS A SECURITY AUDIT REPORT DOCUMENT WHICH MAY CONTAIN INFORMATION WHICH IS CONFIDENTIAL. WHICH INCLUDES ANY POTENTIAL VULNERABILITIES AND MALICIOUS CODES WHICH CAN BE USED TO EXPLOIT THE SOFTWARE. THIS MUST BE REFERRED INTERNALLY AND ONLY SHOULD BE MADE AVAILABLE TO THE PUBLIC AFTER ISSUES ARE RESOLVED.

Email: audit@EtherAuthority.io

Introduction

EtherAuthority was contracted by the AIDFY team to perform the Security audit of the AIDFY Token smart contract code. The audit used manual analysis as well as automated software tools. This report presents all the findings regarding the audit performed on May 28th, 2025.

The purpose of this audit was to address the following:

- Ensure that all claimed functions exist and function correctly.
- Identify any security vulnerabilities that may be present in the smart contract.

Project Background

This is a standard and extensible implementation of an **ERC20 token** written in Solidity, primarily designed for Ethereum-compatible blockchains. The code imports and uses OpenZeppelin's modular and audited contracts.

Major Components:

1. **Context (Abstract Contract):**
    a. Provides utility functions to retrieve the msg.sender and msg.data.
    b. Used internally by contracts to make them more flexible for meta-transactions.
2. **IERC20 (Interface):**
    a. Defines the standard ERC20 functions and events (transfer, approve, transferFrom, allowance, etc.).
    b. Events: Transfer and Approval.
3. **IERC20Metadata (Interface):**
    a. Extends IERC20 with additional metadata functions
        i. name()
        ii. symbol()
        iii. decimals()
4. **ERC20 (Main Contract):**
    a. Implements the full logic of the ERC20 token standard.
    b. Uses Context for compatibility with meta-transactions.

c. Implements the IERC20 and IERC20Metadata interfaces.

Audit scope

Name	**Code Review and Security Analysis Report for AIDFY Token Smart Contract**
Platform	**Ethereum**
Language	**Solidity**
File	AIDoneForYou.sol
Smart Contract Code	0xdca33ab01d167d0a04ebe7fbba055cef83896056
Audit Date	May 28th, 2025

Claimed Smart Contract Features

Claimed Feature Detail	Our Observation
Tokenomics: • Name: AI Done For You • Symbol: AIDFY • Decimals: 18 • Total supply of 500,500 (a little over half a million). • No more token minting.	**YES, this is valid.**
Ownership Control: • There are no owner functions, which makes it 100% decentralized.	**YES, this is valid.**

Audit Summary

According to the standard audit assessment, Customer`s solidity-based smart contracts are **“Secured”**. This token contract does not have any ownership control, hence it is **100% decentralized**.

We used various tools like Slither, Solhint, and Remix IDE. At the same time, this finding is based on a critical analysis of the manual audit.

All issues found during automated analysis were manually reviewed, and applicable vulnerabilities are presented in the Audit overview section. A general overview is presented in the AS-IS section, and all identified issues can be found in the Audit overview section.

We found 0 critical, 0 high, 0 medium and 0 low, and 1 very low level issues.
We confirm that 1 very low-severity issue is acknowledged in the smart contract code.

Investors' Advice: A Technical audit of the smart contract does not guarantee the ethical nature of the project. Any owner-controlled functions should be executed by the owner with responsibility. All investors/users are advised to do their due diligence before investing in the project.

Technical Quick Stats

Main Category	Subcategory	Result
Contract Programming	Solidity version not specified	Passed
	Solidity version is too old	Passed
	Integer overflow/underflow	Passed
	Function input parameters lack of check	Passed
	Function input parameters check bypass	Passed
	Function access control lacks management	Passed
	Critical operation lacks an event log	Passed
	Human/contract checks bypass	Passed
	Random number generation/use vulnerability	N/A
	Fallback function misuse	Passed
	Race condition	Passed
	Logical vulnerability	Passed
	Features claimed	Passed
	Other programming issues	Passed
Code Specification	Function visibility not explicitly declared	Passed
	Var. storage location not explicitly declared	Passed
	Use keywords/functions to be deprecated	Passed
	Unused code	Moderated
Gas Optimization	“Out of Gas” Issue	Passed
	High consumption ‘for/while’ loop	Passed
	High consumption ‘storage’ storage	Passed
	Assert() misuse	Passed
Business Risk	The maximum limit for mintage not set	Passed
	“Short Address” Attack	Passed
	“Double Spend” Attack	Passed

Overall Audit Result: **PASSED**

Business Risk Analysis

Category	Result
Buy Tax	0%
Sell Tax	0%
Cannot Buy	No
Cannot Sell	No
Max Tax	0%
Modify Tax	No
Fee Check	Not Detected
Is Honeypot	Not Detected
Trading Cooldown	Not Detected
Can Pause Trade?	Not Detected
Pause Transfer?	No
Max Tax?	No
Is it Anti-whale?	Not Detected
Is Anti-bot?	Not Detected
Is it a Blacklist?	No
Blacklist Check	No
Can Mint?	No
Is it Proxy?	No
Can Take Ownership?	No
Hidden Owner?	Not Detected
Self Destruction?	Not Detected
Auditor Confidence	High

Overall Audit Result: **PASSED**

Email: audit@EtherAuthority.io

Code Quality

This audit scope has 1 smart contract. Smart contract contains Libraries, Smart contracts, inheritance, and Interfaces. This is a compact and well-written smart contract.

The libraries in AIDFY Token are part of its logical algorithm. A library is a different type of smart contract that contains reusable code. Once deployed on the blockchain (only once), it is assigned a specific address, and its properties/methods can be reused many times by other contracts in the AIDFY Token.

The AIDFY team has not provided scenario and unit test scripts, which would have helped to determine the integrity of the code in an automated way.

Code parts are well commented on in the smart contracts. Ethereum's NatSpec commenting style is recommended.

Documentation

We were given an AIDFY Token smart contract code in the form of an Etherscan web link.

As mentioned above, code parts are well commented on. And the logic is straightforward. So it is easy to quickly understand the programming flow as well as complex code logic. Comments are very helpful in understanding the overall architecture of the protocol.

Use of Dependencies

As per our observation, the libraries used in this smart contract infrastructure are based on well-known industry-standard open-source projects.

Apart from libraries, its functions are not used in external smart contract calls.

AS-IS overview

AIDoneForYou.sol

Functions

Sl.	Functions	Type	Observation	Conclusion
1	constructor	read	Passed	No Issue
2	name	read	Passed	No Issue
3	symbol	read	Passed	No Issue
4	decimals	read	Passed	No Issue
5	totalSupply	read	Passed	No Issue
6	balanceOf	read	Passed	No Issue
7	transfer	write	Passed	No Issue
8	allowance	read	Passed	No Issue
9	approve	write	Passed	No Issue
10	transferFrom	write	Passed	No Issue
11	increaseAllowance	write	Passed	No Issue
12	decreaseAllowance	write	Passed	No Issue
13	_transfer	internal	Passed	No Issue
14	_mint	internal	Passed	No Issue
15	_burn	internal	Unused internal function	Refer to Audit Findings
16	_approve	internal	Passed	No Issue
17	_spendAllowance	internal	Passed	No Issue
18	_beforeTokenTransfer	internal	Passed	No Issue
19	_afterTokenTransfer	internal	Passed	No Issue
20	_msgSender	internal	Passed	No Issue
21	_msgData	internal	Unused internal function	Refer to Audit Findings

Severity Definitions

Risk Level	Description
Critical	Critical vulnerabilities are usually straightforward to exploit and can lead to token loss, etc.
High	High-level vulnerabilities are difficult to exploit; however, they also have a significant impact on smart contract execution, e.g., public access to crucial
Medium	Medium-level vulnerabilities are important to fix; however, they can't lead to tokens lose.
Low	Low-level vulnerabilities are mostly related to outdated, unused, etc., code snippets, which can't have a significant impact on execution
Lowest / Code Style / Best Practice	Lowest-level vulnerabilities, code style violations, and info statements can't affect smart contract execution and can be ignored.

Audit Findings

Critical Severity

No Critical severity vulnerabilities were found.

High Severity

No High severity vulnerabilities were found.

Medium

No Medium severity vulnerabilities were found.

Low

No Low severity vulnerabilities were found.

Very Low / Informational / Best practices:

(1) Unused internal function:

These functions are not used in the contract.

context.sol

- _msgData

ERC20.sol

- _burn

Resolution: Remove all these functions from the contract.

Status: This issue is acknowledged and safely ignored because it is part of the OpenZepelin library and does not raise any major security vulnerabilities.

.

Centralization Risk

The AIDFY Token smart contracts do not have any ownership control, **hence it is 100% decentralized.**

Therefore, there is **no** centralization risk.

Conclusion

We were given a contract code in the form of an Etherscan web link. And we have used all possible tests based on the given objects as files. We had observed 1 Informational issue in the smart contracts. We confirm that 1 very low-severity issue is acknowledged in the smart contract code. So, **it's good to go for the production**.

Since possible test cases can be unlimited for such a smart contract protocol, we provide no such guarantee of future outcomes. We have used all the latest static tools and manual observations to cover the maximum possible test cases to scan everything.

Smart contracts within the scope were manually reviewed and analyzed with static analysis tools. Smart Contract's high-level description of functionality was presented in the As-is overview section of the report.

The audit report contains all found security vulnerabilities and other issues in the reviewed code.

Security state of the reviewed smart contract, based on the standard audit procedure scope, is **"Secured".**

Our Methodology

We like to work with a transparent process and make our reviews a collaborative effort. The goals of our security audits are to improve the quality of systems we review and aim for sufficient remediation to help protect users. The following is the methodology we use in our security audit process.

Manual Code Review:

In manually reviewing all of the code, we look for any potential issues with code logic, error handling, protocol and header parsing, cryptographic errors, and random number generators. We also watch for areas where more defensive programming could reduce the risk of future mistakes and speed up future audits. Although our primary focus is on the in-scope code, we examine dependency code and behavior when it is relevant to a particular line of investigation.

Vulnerability Analysis:

Our audit techniques included manual code analysis, user interface interaction, and whitebox penetration testing. We look at the project's website to get a high-level understanding of what functionality the software under review provides. We then meet with the developers to gain an appreciation of their vision of the software. We install and use the relevant software, exploring the user interactions and roles. While we do this, we brainstorm threat models and attack surfaces. We read design documentation, review other audit results, search for similar projects, examine source code dependencies, skim open issue tickets, and generally investigate details other than the implementation.

Documenting Results:

We follow a conservative, transparent process for analyzing potential security vulnerabilities and seeing them through successful remediation. Whenever a potential issue is discovered, we immediately create an Issue entry for it in this document, even though we have not yet verified the feasibility and impact of the issue. This process is conservative because we document our suspicions early, even if they are later shown to not represent exploitable vulnerabilities. We generally follow a process of first documenting the suspicion with unresolved questions, then confirming the issue through code analysis, live experimentation, or automated tests. Code analysis is the most tentative, and we strive to provide test code, log captures, or screenshots demonstrating our confirmation. After this, we analyze the feasibility of an attack in a live system.

Suggested Solutions:

We search for immediate mitigations that live deployments can take, and finally, we suggest the requirements for remediation engineering for future releases. The mitigation and remediation recommendations should be scrutinized by the developers and deployment engineers, and successful mitigation and remediation are an ongoing collaborative process after we deliver our report, and before the details are made public.

Disclaimers

EtherAuthority.io Disclaimer

EtherAuthority team has analyzed this smart contract by the best industry practices at the date of this report, about: 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).

Because the total number of test cases is unlimited, the audit makes no statements or warranties on the security of the code. It also 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 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 alone. We also suggest conducting a bug bounty program to confirm the high level of security of this smart contract.

Technical Disclaimer

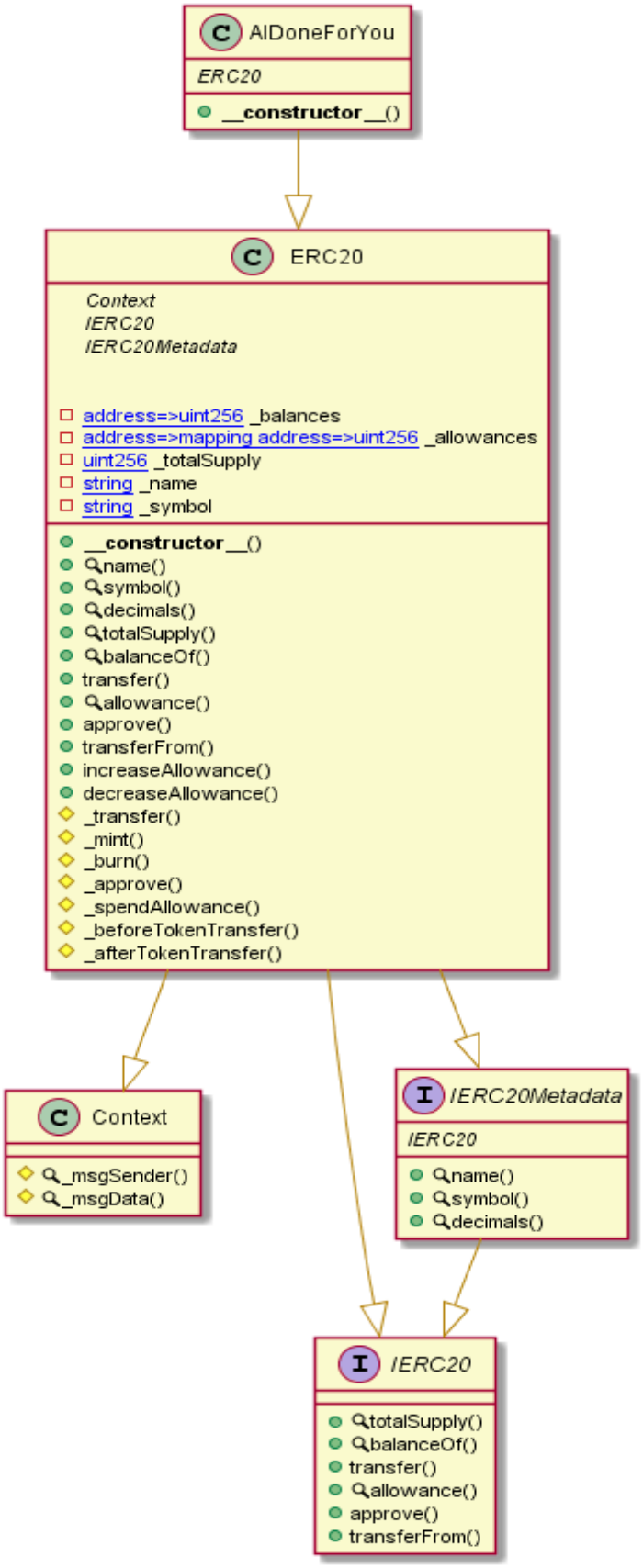
Smart contracts are deployed and executed on the blockchain platform. The platform, its programming language, and other software related to the smart contract can have their own vulnerabilities that can lead to hacks. Thus, the audit can't guarantee the explicit security of the audited smart contracts.

Email: audit@EtherAuthority.io

Appendix

Code Flow Diagram - AI Done For You

AIDFY Token



Email: audit@EtherAuthority.io

Slither Results Log

Slither is a Solidity static analysis framework that uses vulnerability detectors, displays contract details, and provides an API for writing custom analyses. It helps developers identify vulnerabilities, improve code comprehension, and prototype custom analyses quickly. The analysis includes a report with warnings and errors, allowing developers to quickly prototype and fix issues.

We analyzed the project together. Below are the results.

Slither Log >> AIDoneForYou.sol

```
INFO:Detectors:
AIDoneForYou.constructor().totalSupply (AIDoneForYou.sol#550) shadows:
	- ERC20.totalSupply() (AIDoneForYou.sol#245-247) (function)
	- IERC20.totalSupply() (AIDoneForYou.sol#69) (function)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#local-variable-shadowing
INFO:Detectors:
Context._msgData() (AIDoneForYou.sol#37-39) is never used and should be removed
ERC20._burn(address,uint256) (AIDoneForYou.sol#436-452) is never used and should be removed
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#dead-code
INFO:Slither:AIDoneForYou.sol analyzed (5 contracts with 93 detectors), 3 result(s) found
```

Solidity Static Analysis

Static code analysis is used to identify many common coding problems before a program is released. It involves examining the code manually or using tools to automate the process. Static code analysis tools can automatically scan the code without executing it.

AIDoneForYou.sol

```
Gas costs:
Gas requirement of function ERC20.increaseAllowance is infinite: If the gas requirement of a function is higher than the block gas limit, it cannot be executed. Please avoid loops in your functions or actions that modify large areas of storage (this includes clearing or copying arrays in storage)
Pos: 355:34:

Gas costs:
Gas requirement of function AIDoneForYou.decreaseAllowance is infinite: If the gas requirement of a function is higher than the block gas limit, it cannot be executed. Please avoid loops in your functions or actions that modify large areas of storage (this includes clearing or copying arrays in storage)
Pos: 380:13:

Guard conditions:
Use "assert(x)" if you never ever want x to be false, not in any circumstance (apart from a bug in your code). Use "require(x)" if x can be false, due to e.g. invalid input or a failing external component.
Pos: 522:49:
```

Solhint Linter

Linters are the utility tools that analyze the given source code and report programming errors, bugs, and stylistic errors. For the Solidity language, there are some linter tools available that a developer can use to improve the quality of their Solidity contracts.

AIDoneForYou.sol

```
Compiler version 0.8.30 does not satisfy the ^0.5.8 semver
requirement
Pos: 1:19
Explicitly mark visibility in function (Set ignoreConstructors to
true if using solidity >=0.7.0)
Pos: 5:204
Error message for require is too long
Pos: 9:354
Error message for require is too long
Pos: 9:381
Error message for require is too long
Pos: 9:382
Error message for require is too long
Pos: 9:387
Error message for require is too long
Pos: 9:436
Error message for require is too long
Pos: 9:441
Error message for require is too long
Pos: 9:471
Error message for require is too long
Pos: 9:472
Code contains empty blocks
Pos: 24:518
Code contains empty blocks
Pos: 24:538
Explicitly mark visibility in function (Set ignoreConstructors to
true if using solidity >=0.7.0)
Pos: 5:547
```

Software analysis result:

These software reported many false positive results and some are informational issues. So, those issues can be safely ignored.

Email: audit@EtherAuthority.io