www.EtherAuthority.io
audit@etherauthority.io

SMART CONTRACT

Security Audit Report

Project: Dai Stablecoin
Website: makerdao.com
Platform: Ethereum
Language: Solidity
Date: March 4th, 2024

Table of contents

`

THIS IS SECURITY AUDIT REPORT DOCUMENT AND 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.

Introduction

As part of EtherAuthority's community smart contracts audit initiatives, the Dai Stablecoin token smart contract from makerdao.com was audited extensively. The audit has been performed using manual analysis as well as using automated software tools. This report presents all the findings regarding the audit performed on March 14th, 2024.

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 Solidity code defines a contract named `Dai` which represents a stablecoin token. Let's break down the code:
    - **LibNote Contract:** This contract provides a `note` modifier that logs events with specific data. It's used to provide additional context for events emitted by other contracts.
    - **Dai Contract:**
        - **Auth Management:** It includes functions `rely` and `deny` for managing authorization of certain addresses.
        - **ERC20 Data:** Defines ERC20 standard data such as `name`, `symbol`, `decimals`, `totalSupply`, `balanceOf`, and `allowance`.
        - **Events:** Emits events for `Approval` and `Transfer`.
        - **EIP712 Niceties:** Precomputes a `DOMAIN_SEPARATOR` for EIP712 signatures.
        - **Constructor:** Initializes the contract with the deploying address as an authorized address.
        - **Token Functions:**
            - **transfer:** Transfers tokens from the sender to a specified address.
            - **transferFrom:** Transfers tokens from a specified address to another address, if allowed.

        - **mint:** Mints new tokens to a specified address, which increases the total supply.
        - **burn:** Burns tokens from a specified address, which decreases the total supply.
        - **approve:** Approves an address to spend tokens on behalf of another address.
    - **Alias Functions:** Provides aliases push, pull, and move for easier token transfer operations.
    - **Permit Function:** Implements the EIP2612 permit function for approvals via signature.

- This contract represents a basic implementation of an ERC20 token with additional features such as permit approvals and authorization management. It also includes safety checks for arithmetic operations and authorization checks for certain functions.

Audit scope

Name	Code Review and Security Analysis Report for Dai Stablecoin Smart Contract
Platform	Ethereum
File	Dai.sol
Smart Contract Code	0x6b175474e89094c44da98b954eedeac495271d0f
Audit Date	March 4th, 2024

Claimed Smart Contract Features

<table>
<tr><th>Claimed Feature Detail</th><th>Our Observation</th></tr>
<tr><td>Tokenomics:
Name: Dai Stablecoin
Symbol: DAI
Decimals: 18
Version: 1</td><td>YES, This is valid.</td></tr>
<tr><td>Auth control:
Mint new tokens.</td><td>YES, This is valid.</td></tr>
</table>

Audit Summary

According to the standard audit assessment, the Customer`s solidity-based smart contracts are **"Secured"**. Also, these contracts contain owner control, which does not make them fully 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. The 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, 0 low and 5 very low level issues.

Investor 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	The solidity version not specified	Passed
	Solidity version is too old	Moderated
	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 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	Moderated
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	Not Detected
Fee Check	No
Is Honeypot	Not Detected
Trading Cooldown	Not Detected
Can Pause Trade?	No
Pause Transfer?	Not Detected
Max Tax?	No
Is it Anti-whale?	Not Detected
Is Anti-bot?	Not Detected
Is it a Blacklist?	Not Detected
Blacklist Check	No
Can Mint?	Yes
Is it a Proxy?	Not Detected
Can Take Ownership?	Not Detected
Hidden Owner?	Not Detected
Self Destruction?	Not Detected
Auditor Confidence	High

Overall Audit Result: PASSED

Code Quality

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

The libraries in Dai Stablecoin 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 Dai coin.

The EtherAuthority team has no scenario and unit test scripts, which would have helped to determine the integrity of the code in an automated way.

Code parts are not well commented on in the smart contracts. Ethereum's NatSpec commenting style is recommended.

Documentation

We were given a Dai coin smart contract code in the form of an Etherscan web link.

As mentioned above, code parts are not well commented on. but 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 that 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

Functions

Sl.	Functions	Type	Observation	Conclusion
1	constructor	write	Passed	No Issue
2	add	internal	Missing error message in required condition	Refer Audit Findings
3	sub	internal	Missing error message in required condition	Refer Audit Findings
4	rely	external	Passed	No Issue
5	deny	external	Passed	No Issue
6	auth	modifier	Passed	No Issue
7	transfer	external	Passed	No Issue
8	transferFrom	write	Passed	No Issue
9	mint	external	Centralized risk	Refer Audit Findings
10	burn	external	Passed	No Issue
11	approve	external	Passed	No Issue
12	push	external	Passed	No Issue
13	pull	external	Passed	No Issue
14	move	external	Passed	No Issue
15	permit	external	Passed	No Issue
16	note	modifier	Passed	No Issue

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, that 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) Use the latest solidity version:

```
pragma solidity =0.5.12;
```

Use the latest solidity version while contract deployment to prevent any compiler version-level bugs.

Resolution: Please use versions greater than 0.8.7.

(2) Variable Naming Convention:

```
// --- EIP712 niceties ---
bytes32 public DOMAIN_SEPARATOR;
```

In the smart contract code, the variable DOMAIN_SEPARATOR does not follow the recommended Solidity naming convention. Solidity convention suggests using mixed case for variable names. While this does not pose a security risk, adhering to naming conventions improves code readability and consistency.

Resolution: It is advisable to rename the DOMAIN_SEPARATOR variable to follow the mixed-case naming convention, which makes the code more consistent with Solidity best practices.

(3) Centralized risk:

```
function mint(address usr, uint wad) external auth {
        balanceOf[usr] = add(balanceOf[usr], wad);
        totalSupply    = add(totalSupply, wad);
        emit Transfer(address(0), usr, wad);
    }
```

Only Auth can mint a token.

Resolution: To make the smart contract 100% decentralized. We suggest renouncing ownership of the smart contract once its function is completed.

(4) Missing error message in required condition:

```
function add(uint x, uint y) internal pure returns (uint z) {
        require((z = x + y) >= x);
    }
function sub(uint x, uint y) internal pure returns (uint z) {
        require((z = x - y) <= x);
    }
```

It is best practice to add custom error messages in every required condition, which would be helpful in debugging as well as giving a clear indication of any transaction failure.

Resolution: Add custom error messages in every required condition.

(5) Unwanted comments:

```
 /* pragma solidity 0.5.12; */
/* import "./lib.sol"; */
```

Unwanted comments found in code.

Resolution: We suggest removing unwanted comments like this.

Centralization

This smart contract has some functions which can be executed by the Admin (Owner) only. If the admin wallet's private key would be compromised, then it would create trouble. The following are Admin functions:

Dai.sol

- mint: Mint tokens by the auth.

To make the smart contract 100% decentralized, we suggest renouncing ownership of the smart contract once its function is completed.

Conclusion

We were given a contract code in the form of Etherscan web links. And we have used all possible tests based on given objects as files. We observed 5 informational issues in the smart contracts. And those issues are not critical. So, **it's good to go for the production**.

Since possible test cases can be unlimited for such smart contracts 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.

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

Security state of the reviewed smart contract, based on 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 the 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 white box 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 is 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 in accordance with the best industry practices at the date of this report, in relation to: 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).

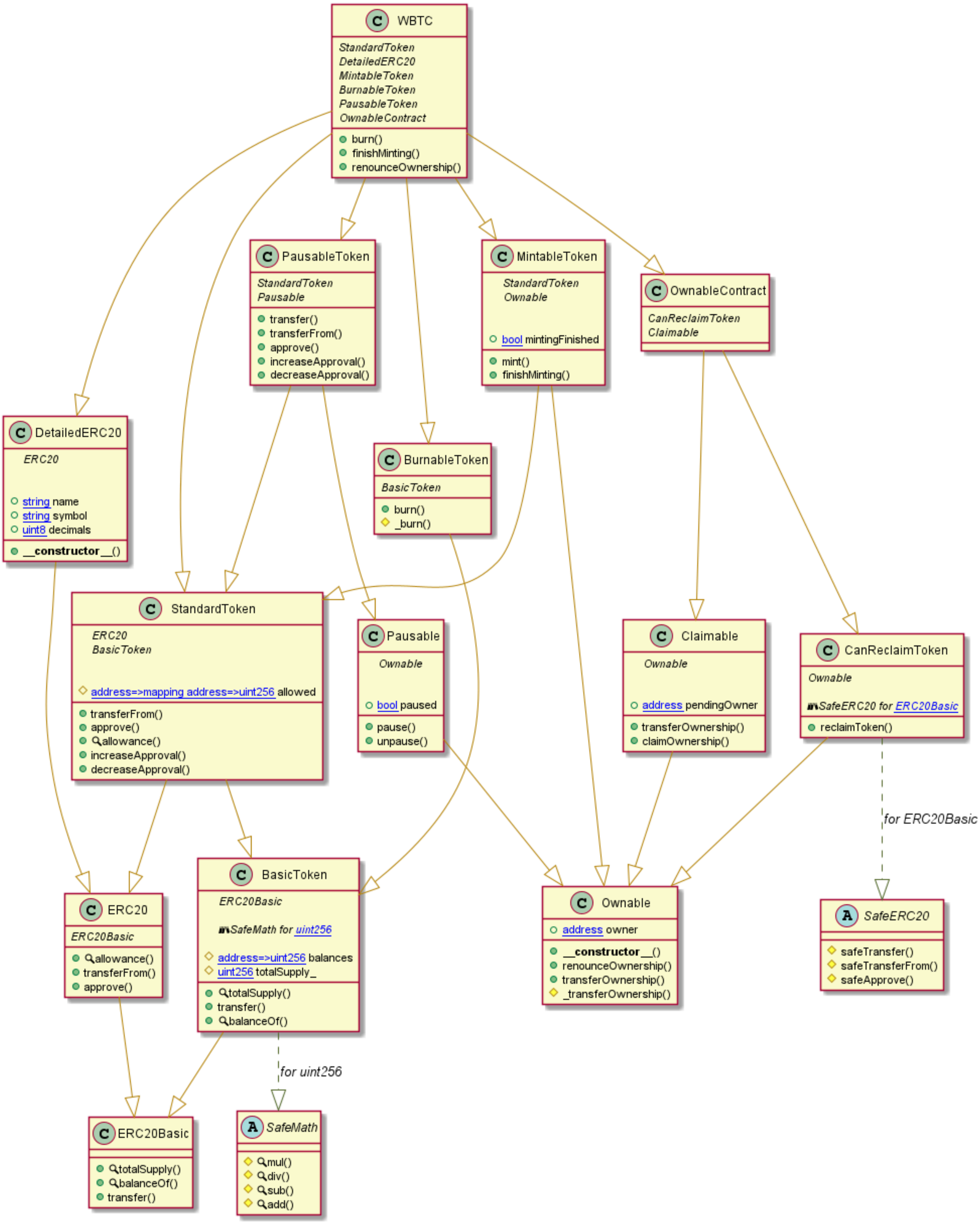
Due to the fact that the total number of test cases are unlimited, the audit makes no statements or warranties on security of the code. It also cannot be considered as a sufficient assessment regarding the utility and safety of the code, bugfree status or any other statements of the contract. While we 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 only. 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 explicit security of the audited smart contracts.

Appendix

Code Flow Diagram - Dai Stablecoin



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 did the analysis of the project altogether. Below are the results.

Slither Log >> Dai.sol

```
Dai.permit(address,address,uint256,uint256,bool,uint8,bytes32,bytes32) (Dai.sol#134-156) uses timestamp for comparisons
        Dangerous comparisons:
        - require(bool,string)(expiry == 0 || now <= expiry,Dai/permit-expired) (Dai.sol#151)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#block-timestamp

Pragma version=0.5.12 (Dai.sol#4) allows old versions
solc-0.5.12 is not recommended for deployment
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#incorrect-versions-of-solidity

Constant Dai.version (Dai.sol#49) is not in UPPER_CASE_WITH_UNDERSCORES
Variable Dai.DOMAIN_SEPARATOR (Dai.sol#69) is not in mixedCase
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#conformance-to-solidity-naming-conventions
Dai.sol analyzed (2 contracts with 84 detectors), 5 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.

Dai.sol

Inline assembly:

The Contract uses inline assembly, this is only advised in rare cases. Additionally static analysis modules do not parse inline Assembly, this can lead to wrong analysis results.
more
Pos: 35:8:

Block timestamp:

Use of "now": "now" does not mean current time. "now" is an alias for "block.timestamp". "block.timestamp" can be influenced by miners to a certain degree, be careful.
more
Pos: 188:31:

Gas costs:

Gas requirement of function Dai.mint 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: 138:4:

Gas costs:

Gas requirement of function Dai.burn 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: 143:4:

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.

more

Pos: 189:8:

Similar variable names:

Dai.mint(address,uint256) : Variables have very similar names "wards" and "wad". Note: Modifiers are currently not considered by this static analysis.

Pos: 141:39:

Similar variable names:

Dai.burn(address,uint256) : Variables have very similar names "wards" and "wad". Note: Modifiers are currently not considered by this static analysis.

Pos: 144:34:

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.

more

Pos: 189:8:

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.

Dai.sol

```
Avoid using inline assembly. It is acceptable only in rare cases
Pos: 9:34
Constant name must be in capitalized SNAKE_CASE
Pos: 5:83
Constant name must be in capitalized SNAKE_CASE
Pos: 5:84
Constant name must be in capitalized SNAKE_CASE
Pos: 5:85
Constant name must be in capitalized SNAKE_CASE
Pos: 5:86
Provide an error message for require
Pos: 9:98
Provide an error message for require
Pos: 9:101
Variable name must be in mixedCase
Pos: 5:105
Avoid making time-based decisions in your business logic
Pos: 32:187
```

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