Ether Authority

www.EtherAuthority.io
audit@etherauthority.io

SMART CONTRACT

Security Audit Report

Project: USD Coin (USDC)
Website: circle.com/usdc
Platform: Polygon
Language: Solidity
Date: April 7th, 2025

Table of Contents

Email: audit@EtherAuthority.io

THIS IS A SECURITY AUDIT REPORT DOCUMENT THAT MAY CONTAIN INFORMATION THAT 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

As part of EtherAuthority's community smart contract audit initiatives, the smart contract of the USDC Token from circle.com/usdc was audited. The audit was performed using manual analysis and automated software tools. This report presents all the findings regarding the audit performed on April 7th, 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

FiatTokenV2_2 is an **upgradeable ERC20-compliant smart contract** designed to represent fiat-backed stablecoins with advanced features for security, flexibility, and interoperability. It builds on previous versions (FiatTokenV1, V2, and V2_1) and introduces new capabilities while preserving backward compatibility.

Key Features:

Upgradeable Architecture

- Uses a versioned initialize and initializeV2, initializeV2_1, initializeV2_2 pattern to support upgrades without storage clashes.
- Complies with upgradeability patterns (like OpenZeppelin's Initializable) without using constructors.

Blacklist Mechanism

- Efficient **bit-packed blacklist status** in balanceAndBlacklistStates mapping:
  - Highest bit indicates blacklist status.
  - The lower 255 bits hold the actual balance.
- Allows regulatory compliance by freezing blacklisted accounts.

EIP-2612: Permit Functionality

- Allows token approvals via **off-chain signatures**, saving users gas costs on

approval transactions.

- Supports both traditional signature components (v, r, s) and packed bytes signatures.

EIP-3009: Meta-Transactions

- Enables **off-chain signed transfers** via:
  - transferWithAuthorization
  - receiveWithAuthorization
  - cancelAuthorization
- Great for gasless token interactions in wallets and dApps.

Domain Separation and Authorization

- Implements EIP-712-style **domain separator logic** to protect against replay attacks across chains.
- Prevents replay of permit and authorization messages by storing used nonces and authorization states.

Audit scope

Name	**Code Review and Security Analysis Report for USD Coin (USDC) Token Smart Contract**
Platform	**Polygon**
File 1	FiatTokenV2_2.sol
File 1 Smart Contract	0x235ae97b28466db30469b89a9fe4cff0659f82cb
File 2	FiatTokenProxy.sol
File 2 Smart Contract	0x3c499c542cEF5E3811e1192ce70d8cC03d5c3359
Audit Date	April 7th, 2025

Claimed Smart Contract Features

Claimed Feature Detail	Our Observation
Key Features: **1. ERC20 Core Functionality** ● Standard ERC20 interface: ○ transfer, approve, transferFrom, allowance, balanceOf, totalSupply ● Extended with internal balance structure for blacklisting logic. **2. Blacklist Mechanism** ● Efficient **bitwise blacklist** system using a packed uint256: ○ balanceAndBlacklistStates[address] stores both balance and blacklist flag. ○ Most significant bit (MSB) marks blacklist status. ● Affects all token operations (transfer, transferFrom, etc.). ● Admin functions to blacklist and unBlacklist addresses. ● Supports regulatory compliance and asset freezing. **3. EIP-2612: Permit (Gasless Approvals)** ● permit(address owner, address spender, uint256 value, uint256 deadline, uint8 v, bytes32 r, bytes32 s) ● Off-chain approval via signed message, reducing user gas costs.	**YES, This is valid.**

- Stores nonces to prevent replay attacks.
- Compliant with EIP-712 domain separator logic.

4.EIP-3009: Meta-Transactions (Gasless Transfers)

- Off-chain authorized transfer flow:
  - transferWithAuthorization
  - receiveWithAuthorization
  - cancelAuthorization
- Enables **gasless and delegated transfers** by signed authorization.
- Protects against replay via:
  - authorizationStates[from][nonce] mapping
  - Authorization validity windows (validAfter / validBefore)

5. Upgradeable Initialization

- Follows versioned initializer functions:
  - initialize, initializeV2, initializeV2_1, initializeV2_2
- Allows safe upgrades without breaking existing storage layout or logic.
- Respects OpenZeppelin's Initializable pattern.

6. Authorization Replay Protection

- Nonces used for both EIP-2612 and EIP-3009 to prevent message re-use.
- authorizationStates mapping tracks used and unused authorizations.

7. Version Identification

• version() function returns "2.2" to identify the deployed contract version.
• Useful for frontends, off-chain services, and multi-version deployments.
 	
8. Packed Storage Optimization
 	
• Combines blacklist status and token balance into a single storage slot.
• Gas-efficient reads/writes and minimizes state bloat.
• Internal utility functions:
  ○ _isBlacklisted(address)
  ○ _balanceOf(address)
  ○ _setBalance(address, uint256)
  ○ _setBlacklisted(address, bool)
 	
9. Access Control (Inherited)
 	
• Admin-only functions (like mint, blacklist) guarded via access roles.
• Controlled via AccessControl patterns (from earlier versions).	

Audit Summary

According to the standard audit assessment, the Customer's solidity-based smart contract is **"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. 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 1 very low-level issue.

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 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 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	Passed
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 is not set	Passed
	“Short Address” Attack	Passed
	“Double Spend” Attack	Passed

Overall Audit Result: **PASSED**

Code Quality

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

The libraries in USDC 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 USDC Token.

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 well commented on in the smart contract. Ethereum's NatSpec commenting style is recommended.

Documentation

We were given a USDC Token smart contract code in the form of a polygonscan 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 that is 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

FiatTokenV2_2.sol : Functions

Sl.	Functions	Type	Observation	Conclusion
1	version	external	Passed	No Issue
2	initializeV2_2	external	Unprotected Initialize Functions	Refer Audit Findings
3	_chainId	internal	Passed	No Issue
4	_domainSeparator	internal	Passed	No Issue
5	permit	external	whenNotPaused	No Issue
6	transferWithAuthorization	external	notBlacklisted	No Issue
7	receiveWithAuthorization	external	notBlacklisted	No Issue
8	cancelAuthorization	external	cancelAuthorization	No Issue
9	_setBlacklistState	internal	Passed	No Issue
10	_setBalance	internal	Passed	No Issue
11	_isBlacklisted	internal	Passed	No Issue
12	_balanceOf	internal	Passed	No Issue
13	approve	external	Passed	No Issue
14	permit	external	whenNotPaused	No Issue
15	increaseAllowance	external	whenNotPaused	No Issue
16	decreaseAllowance	external	whenNotPaused	No Issue
17	initializeV2_1	external	Unprotected Initialize Functions	Refer 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 being lost
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) Unprotected Initialize Functions:

The initialize functions (initializeV2, initializeV2_1, initializeV2_2) lack proper access control.

Resolution: Add the onlyOwner modifier to these functions to restrict initialization to authorized parties.

Centralization Risk

The USDC Token smart contract does 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 a polygonscan web link. We have used all possible tests based on the given objects as files. We observed 1 informational issue in the smart contract, and those issues are not critical. So, **it's good to go for production**.

Since possible test cases can be unlimited for such smart contract protocols, 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.

The 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 not to 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 as 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 only. We also suggest conducting a bug bounty program to confirm the high level of security of this smart contract.

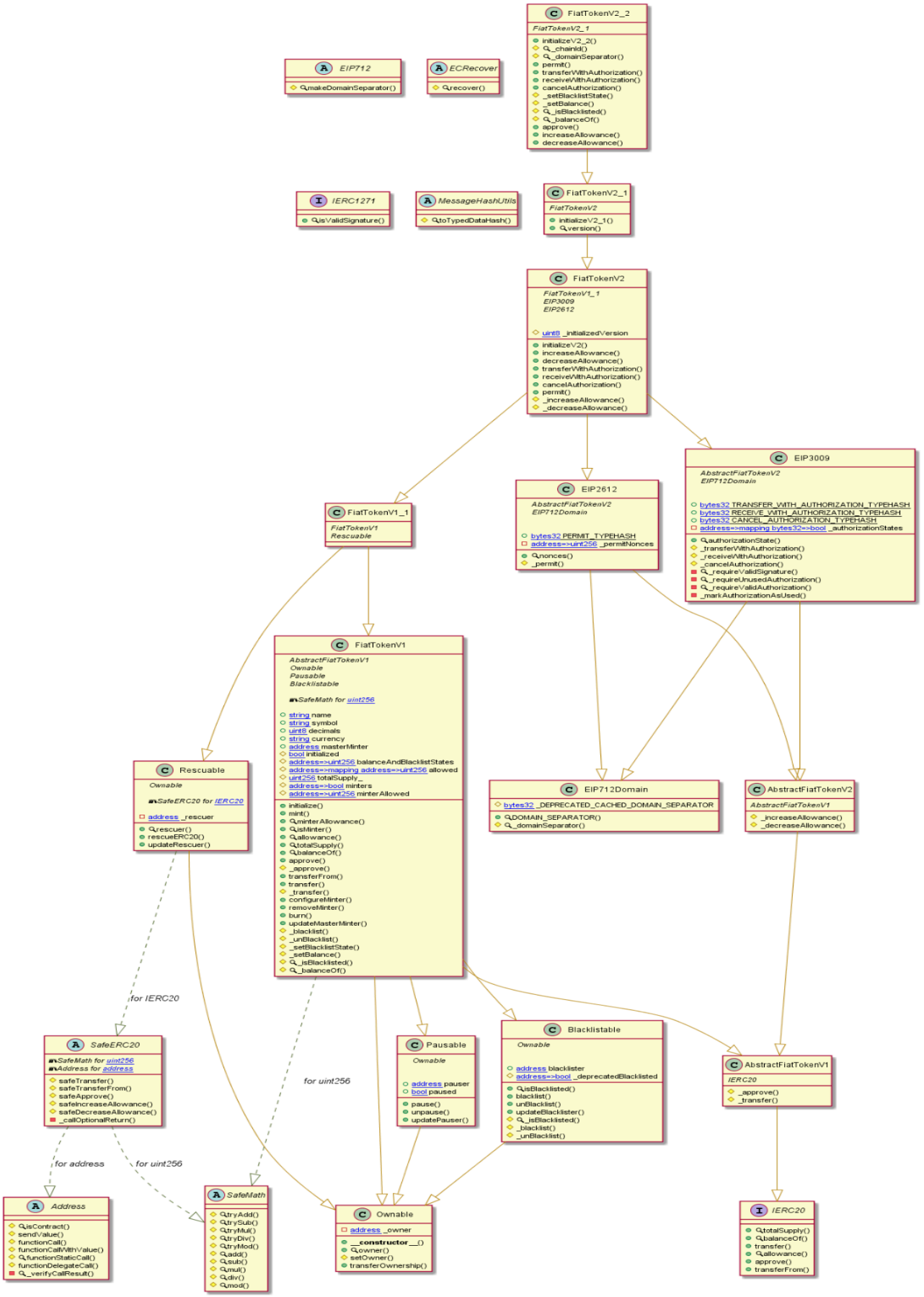
Technical Disclaimer

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.

Email: audit@EtherAuthority.io

Appendix

Code Flow Diagram - USD Coin (USDC) Token



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 >> FiatTokenV2_2.sol

```
INFO:Detectors:
FiatTokenV2_2.permit(address,address,uint256,uint256,uint8,bytes32,bytes32).owner
(FiatTokenV2_2.sol#2278) shadows:
	- Ownable.owner() (FiatTokenV2_2.sol#725-727) (function)
Reference:
https://github.com/crytic/slither/wiki/Detector-Documentation#local-variable-shadowing
INFO:Detectors:
FiatTokenV2_2._chainId() (FiatTokenV2_2.sol#2076-2082) uses assembly
	- INLINE ASM (FiatTokenV2_2.sol#2078-2080)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#assembly-usage
INFO:Detectors:
FiatTokenV2_2.initializeV2_2(address[],string) (FiatTokenV2_2.sol#2046-2070) has costly
operations inside a loop:
	- delete _deprecatedBlacklisted[accountsToBlacklist[i]] (FiatTokenV2_2.sol#2064)
Reference:
https://github.com/crytic/slither/wiki/Detector-Documentation#costly-operations-inside-a-loop
INFO:Detectors:
EIP712Domain._domainSeparator() (FiatTokenV2_2.sol#668-670) is never used and should be
removed
FiatTokenV1._balanceOf(address) (FiatTokenV2_2.sol#1391-1398) is never used and should be
removed
FiatTokenV1._isBlacklisted(address) (FiatTokenV2_2.sol#1376-1384) is never used and should
be removed
FiatTokenV1._setBalance(address,uint256) (FiatTokenV2_2.sol#1369-1371) is never used and
should be removed
FiatTokenV1._setBlacklistState(address,bool) (FiatTokenV2_2.sol#1357-1362) is never used and
should be removed
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#dead-code
INFO:Detectors:
Version constraint >=0.6.0<0.8.0 is too complex.
It is used by:
```

```
        - >=0.6.0<0.8.0 (FiatTokenV2_2.sol#3)
solc-0.6.12 is an outdated solc version. Use a more recent version (at least 0.8.0), if possible.
Reference:
https://github.com/crytic/slither/wiki/Detector-Documentation#incorrect-versions-of-solidity
INFO:Detectors:
Function FiatTokenV2_1.initializeV2_1(address) (FiatTokenV2_2.sol#2017-2028) is not in
mixedCase
Contract FiatTokenV2_2 (FiatTokenV2_2.sol#2039-2315) is not in CapWords
Function FiatTokenV2_2.initializeV2_2(address[],string) (FiatTokenV2_2.sol#2046-2070) is not
in mixedCase
Reference:
https://github.com/crytic/slither/wiki/Detector-Documentation#conformance-to-solidity-naming-c
onventions
INFO:Slither:FiatTokenV2_2.sol analyzed (22 contracts with 93 detectors), 43 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.

FiatTokenV2_2.sol

```
Check-effects-interaction:
Potential violation of Checks-Effects-Interaction pattern in SafeERC20.safeDecreaseAllowance(contract IERC20,address,uint256): Could potentially lead to re-entrancy vulnerability. Note: Modifiers are currently not considered by this static analysis.
Pos: 625:7:

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.
Pos: 2078:11:

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.
Pos: 1777:60:

Gas costs:
Gas requirement of function FiatTokenV1.pause 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: 787:7:

Gas costs:
Gas requirement of function FiatTokenV2_2.permit 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: 2100:7:

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: 1316:11:
```

Solhint Linter

Solhint 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.

FiatTokenV2_2.sol

```
Compiler version >=0.6.0 <0.8.0 does not satisfy the ^0.5.8 semver
requirement
Pos: 1:2
Avoid to use inline assembly. It is acceptable only in rare cases
Pos: 9:112
Code contains empty blocks
Pos: 1:1401
Contract name must be in CamelCase
Pos: 1:1401
Error message for require is too long
Pos: 9:1592
Code contains empty blocks
Pos: 20:1667
Avoid making time-based decisions in your business logic
Pos: 13:1700
Error message for require is too long
Pos: 9:1703
Avoid making time-based decisions in your business logic
Pos: 17:1703
Avoid making time-based decisions in your business logic
Pos: 58:1776
Variable "signature" is unused+
Pos: 9:1773
Variable "typedDataHash" is unused
Pos: 9:1780
Function name must be in mixedCase
Pos: 5:2045
Error message for require is too long
Pos: 13:2058
Avoid to use inline assembly. It is acceptable only in rare cases
Pos: 9:2077
Error message for require is too long
Pos: 9:2223
```

Software analysis result:

This software reported many false positive results, some of which are informational issues. Therefore, those issues can be safely ignored.

Email: audit@EtherAuthority.io