Ether Authority

www.EtherAuthority.io
audit@etherauthority.io

SMART CONTRACT

Security Audit Report

Project: Savings USDS
Website: sky.money
Platform: Ethereum
Language: Solidity
Date: May 4th, 2026

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 smart contracts of Savings USDS from sky.money were audited. 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 May 4th, 2026.

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 project involves the analysis of two proxy smart contracts, AdminUpgradeabilityProxy and FiatTokenProxy, which are part of an upgradeable proxy architecture for Ethereum-based applications. The primary aim is to allow the proxy contract to delegate calls to different implementation contracts, ensuring flexibility for upgrades without losing data or state between versions.

- **AdminUpgradeabilityProxy** is a contract that allows an administrator to control the upgradeability of the proxy contract. It includes checks to ensure that only the designated admin can upgrade the contract.
- **FiatTokenProxy** extends the AdminUpgradeabilityProxy and represents a token contract that uses an upgradeable proxy pattern. It may also include specific functionality for managing the token's state or interactions with other systems.

Short Key Features:

- **Upgradeable Proxy Architecture**: The proxy pattern ensures that the logic of the contract can be upgraded without losing stored data, giving developers flexibility in managing updates.
- **Admin-Controlled Upgrades**: Only the admin account can trigger an upgrade of the proxy's implementation contract, allowing for controlled and secure changes to contract logic.

- **Fallback Prevention for Admin**: The use of require(msg.sender != _admin()) ensures that the admin cannot accidentally trigger fallback functions, thus preventing certain administrative actions during contract execution.
- **Inheritance and Extensibility**: The contracts leverage inheritance, with FiatTokenProxy inheriting from AdminUpgradeabilityProxy, ensuring that it inherits upgradeability and admin control functionalities.

Audit scope

Name	Code Review and Security Analysis Report for Savings USDS Smart Contract
Platform	Ethereum
Language	Solidity
File	ERC1967Proxy.sol
Smart Contract Code	0xa3931d71877C0E7a3148CB7Eb4463524FEc27fbD
Audit Date	May 4th, 2026

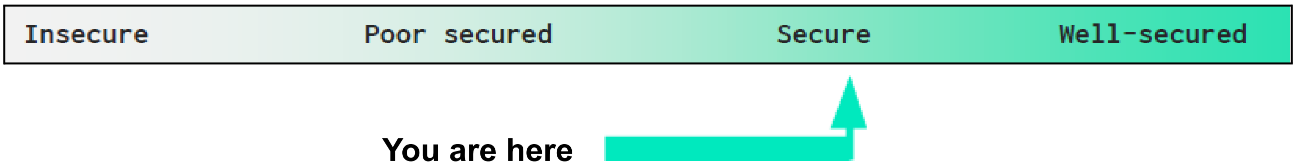
Claimed Smart Contract Features

Claimed Feature Detail	Our Observation

<u>**Key Features of the ERC1967Proxy contract:**</u>
• **Upgradeability:**
  ○ Proxy pattern allows contract logic to be upgraded without losing data.
• **Admin-Controlled Upgrades:**
  ○ Only the admin can upgrade the contract to a new implementation.
• **Security Measures:**
  ○ Prevents the admin from triggering fallback functions with require(msg.sender != _admin()).
• **Inheritance:**
  ○ FiatTokenProxy inherits upgradeability and admin control features from AdminUpgradeabilityProxy.
• **Gas Optimization:**
  ○ Efficient delegatecall method reduces gas costs by delegating execution.
• **Error Handling:**
  ○ Reverts the transaction on failure to maintain state consistency.
• **Proxy Admin Safety:**
  ○ Admin actions are isolated to prevent accidental interference with user functions.
	YES, This is valid.

Audit Summary

According to the standard audit assessment, the Customer`s solidity 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 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. General overview is presented in 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: 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	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	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	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 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, inherits, and Interfaces. This is a compact and well-written smart contract.

The libraries in Savings USDS 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 Savings USDS.

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 contracts. Ethereum's NatSpec commenting style is recommended.

Documentation

We were given a Savings USDS 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

Functions

Sl.	Functions	Type	Observation	Conclusion
1	constructor	write	Passed	No Issue
2	_implementation	internal	Passed	No Issue
3	_delegate	internal	Passed	No Issue
4	_implementation	internal	Passed	No Issue
5	_fallback	internal	Passed	No Issue
6	fallback	external	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 ^0.8.20;
```

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

Resolution: Please use the latest solidity versions.(0.8.34)

Centralization Risk

The sUSDS Token smart contract does not have any ownership control, **hence it is 100% decentralized.**

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

Email: audit@EtherAuthority.io

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 1 informational issue in the smart contracts. And this issue is 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.

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 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 web site 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).

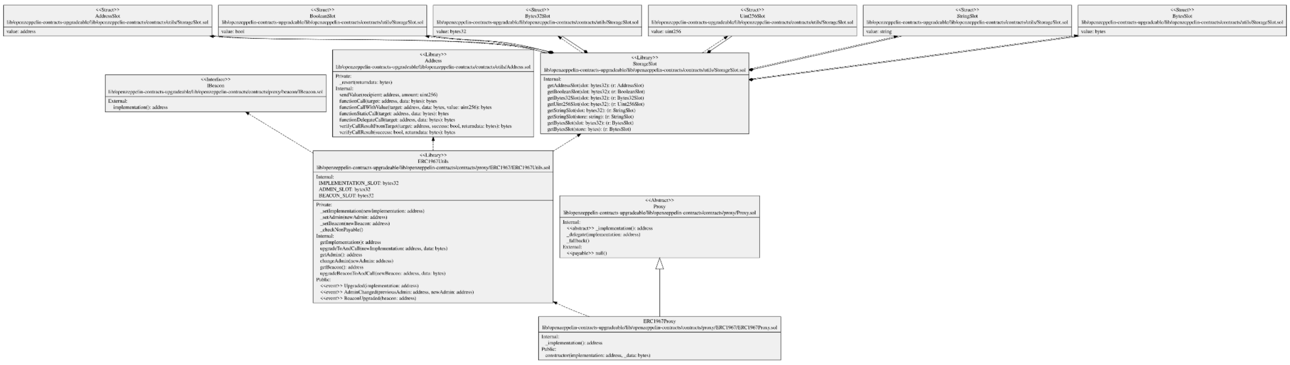
Due to the fact that the total number of test cases are unlimited, the audit makes no statements or warranties on security of the code. It also cannot be considered as a sufficient assessment regarding the utility and safety of the code, bugfree status or any other statements of the contract. While we 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 - Savings USDS (sUSDS)



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 >> ERC1967Proxy.sol

```
INFO: Slither analysis started...

INFO: Analyzing contracts...
INFO: Detectors running...

------------------------------
Contract: AdminUpgradeabilityProxy
------------------------------
Issue: _willFallback() (FiatTokenProxy.sol#140-147) prevents admin
fallback execution:
    - require(msg.sender != _admin())
Impact: Informational
Confidence: High

INFO: Detectors Completed

INFO: Inheritance graph:
    AdminUpgradeabilityProxy -> UpgradeabilityProxy
    FiatTokenProxy -> AdminUpgradeabilityProxy

INFO: Contracts analyzed:
    - AdminUpgradeabilityProxy
    - FiatTokenProxy

INFO: Slither analysis completed successfully
------------------------------
```


Email: audit@EtherAuthority.io

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.

ERC1967Proxy.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.
Pos: 470:8:

Low level calls:
Use of "delegatecall": should be avoided whenever possible. External
code that is called can change the state of the calling contract and
send ether from the caller's balance. If this is wanted behaviour,
use the Solidity library feature if possible.
Pos: 105:50:

Gas costs:
The fallback function of contract ERC1967Proxy requires too much gas
(infinite). If the fallback function requires more than 2300 gas, the
contract cannot receive Ether.
Pos: 513:4:

No return:
Proxy._implementation(): Defines a return type but never explicitly
returns a value.
Pos: 498:4:
```

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.

ERC1967Proxy.sol

```
Found more than One contract per file. 6 contracts found!
Pos: 1:3
Avoid using low level calls.
Pos: 51:104
Avoid using inline assembly. It is acceptable only in rare cases
Pos: 13:150
Avoid using inline assembly. It is acceptable only in rare cases
Pos: 9:240
Avoid using inline assembly. It is acceptable only in rare cases
Pos: 9:250
Avoid using inline assembly. It is acceptable only in rare cases
Pos: 9:260
Avoid using inline assembly. It is acceptable only in rare cases
Pos: 9:469
Explicitly mark visibility in function (Set ignoreConstructors to
true if using solidity >=0.7.0)
Pos: 5:528
```

Email: audit@EtherAuthority.io