Ether Authority

www.EtherAuthority.io
audit@etherauthority.io

SMART CONTRACT

Security Audit Report

Project:	Decentralized USD
Website:	usdd.io
Platform:	Ethereum
Language:	Solidity
Date:	April 28th, 2024

Table of contents

`

Introduction

As part of EtherAuthority's community smart contracts audit initiatives, the smart contracts of Decentralized USD Token from usdd.io 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 April 28th, 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

- Decentralized USD token contract for an ERC20 token called USDD. Here's a breakdown of the contract:
  - **SafeMath Library:** This library provides arithmetic functions with safety checks to prevent overflow and underflow.
  - **IERC20 Interface:** This interface defines the standard functions and events for ERC20 tokens.
  - **Address Library:** This library provides functions to interact with addresses, including checking if an address is a contract and performing low-level calls.
  - **EnumerableSet Library:** This library provides data structures and functions for managing enumerable sets of data.
  - **ContextMixin Contract:** This contract provides context functions for retrieving the sender of a message.
  - **IMintableERC20 Interface:** This interface extends the ERC20 interface with a `mint` function to mint new tokens.
  - **Initializable Contract:** This contract ensures that certain functions can only be called during initialization.
  - **ERC20 Contract:** This is the main ERC20 token contract, implementing the ERC20 interface with additional functionality such as minting, burning, and allowance management.
  - **AccessControl Contract:** This contract provides role-based access control functionality, allowing certain roles to perform specific actions.

- **AccessControlMixin Contract:** This contract is a mix-in for access control, defining modifiers and functions to grant, revoke, and check roles.
- **EIP712Base Contract:** This contract implements the EIP-712 standard for typed structured data hashing and signing.
- **NativeMetaTransaction Contract:** This contract enables meta transactions using EIP-712 structured data.
- **USDD Contract:** This is the actual token contract that inherits from ERC20, AccessControlMixin, ContextMixin, and NativeMetaTransaction. It defines the USDD token, including its constructor and the `mint` function, which can only be called by the `PREDICATE_ROLE`.

- Overall, this contract provides a comprehensive implementation of an ERC20 token with additional features like meta transactions and role-based access control.

Audit scope

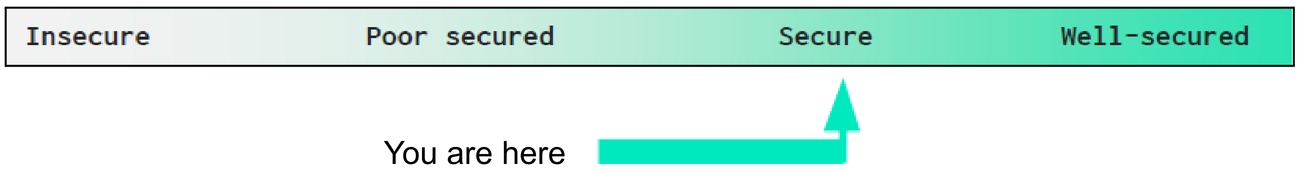
Name	Code Review and Security Analysis Report for Decentralized USD Token Smart Contract
Platform	Ethereum
File	USDD.sol
Ethereum Code	0x0C10bF8FcB7Bf5412187A595ab97a3609160b5c6
Audit Date	April 28th, 2024

Claimed Smart Contract Features

Claimed Feature Detail	Our Observation
Tokenomics: ● Name: Decentralized USD ● Symbol: USDD ● Decimals: 18	**YES, This is valid.**
Admin role control: ● Grants \`role\` to \`account\` can be set by the admin. ● Revokes \`role\` from \`account\` by the admin. ● Renounce Role from \`account\` by the admin.	**YES, This is valid.**

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. 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, 0 low, and 3 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 is not specified	Passed
	The solidity version is too old	Moderated
	Integer overflow/underflow	Passed
	Function input parameters lack 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 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?	No
● Max Tax?	No
● Is it Anti-whale?	No
● Is Anti-bot?	Not Detected
● Is it a Blacklist?	Not Detected
● Blacklist Check	No
● Can Mint?	Yes
● Is it Proxy?	No
● Can Take Ownership?	No
● Hidden Owner?	No
● Self Destruction?	No
● 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 Decentralized USD 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 Decentralized USD 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 contracts. Ethereum's NatSpec commenting style is recommended.

Documentation

We were given a Decentralized USD 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 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	mint	external	Centralized Ownership and Privileges Management	Refer Audit Findings
3	_msgSender	internal	Passed	No Issue
4	executeMetaTransaction	write	Passed	No Issue
5	hashMetaTransaction	internal	Passed	No Issue
6	getNonce	read	Passed	No Issue
7	verify	internal	Passed	No Issue
8	_initializeEIP712	internal	initializer	No Issue
9	_setDomainSeperator	internal	Passed	No Issue
10	getDomainSeperator	read	Passed	No Issue
11	getChainId	write	Passed	No Issue
12	toTypedMessageHash	internal	Passed	No Issue
13	only	modifier	Passed	No Issue
14	_setupContractId	internal	Passed	No Issue
15	hasRole	read	Passed	No Issue
16	getRoleMemberCount	read	Passed	No Issue
17	getRoleMember	read	Passed	No Issue
18	getRoleAdmin	read	Passed	No Issue
19	grantRole	write	Access only admin role	No Issue
20	revokeRole	write	Access only admin role	No Issue
21	renounceRole	write	Access only admin role	No Issue
22	_setupRole	internal	Passed	No Issue
23	_setRoleAdmin	internal	Passed	No Issue
24	_grantRole	write	Passed	No Issue
25	_revokeRole	write	Passed	No Issue
26	name	read	Passed	No Issue
27	symbol	read	Passed	No Issue
28	decimals	read	Passed	No Issue
29	totalSupply	read	Passed	No Issue
30	balanceOf	read	Passed	No Issue
31	transfer	write	Passed	No Issue
32	allowance	read	Passed	No Issue
33	approve	write	Passed	No Issue
34	transferFrom	write	Passed	No Issue
35	increaseAllowance	write	Passed	No Issue
36	decreaseAllowance	write	Passed	No Issue
37	_transfer	internal	Passed	No Issue
38	_mint	internal	Passed	No Issue
39	_burn	internal	Passed	No Issue
40	_approve	internal	Passed	No Issue

41	_setupDecimals	internal	Passed	No Issue
42	_beforeTokenTransfer	internal	Passed	No Issue
43	initializer	modifier	Passed	No Issue
44	_msgSender	internal	Passed	No Issue
45	_msgData	internal	Passed	No Issue
46	msgSender	internal	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.

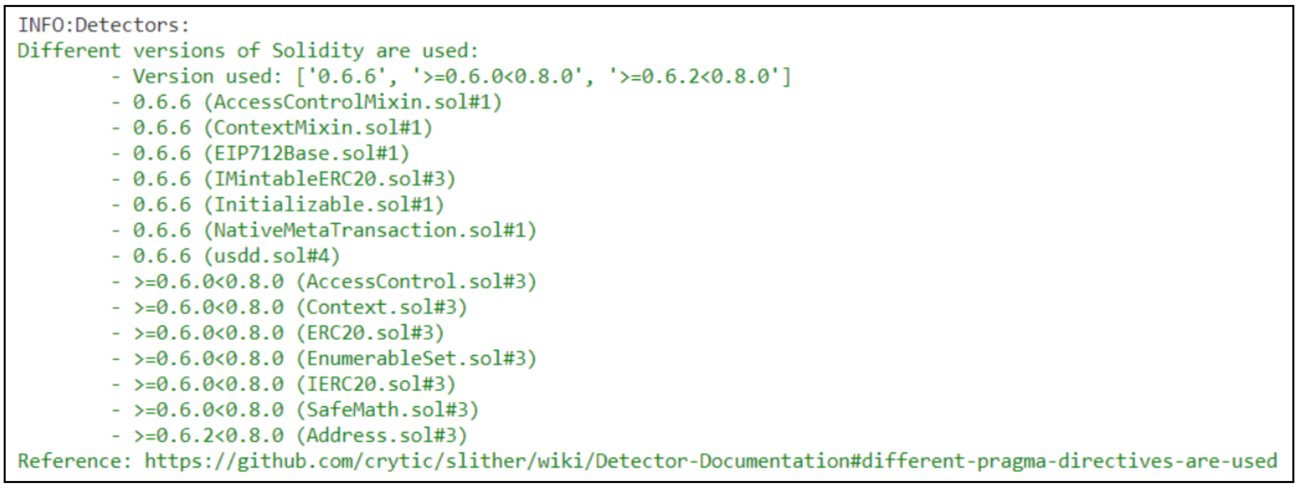
Low

No Low severity vulnerabilities were found.

Very Low / Informational / Best Practices:

(1) Consistent Pragma Solidity Version Usage:

USDD.sol

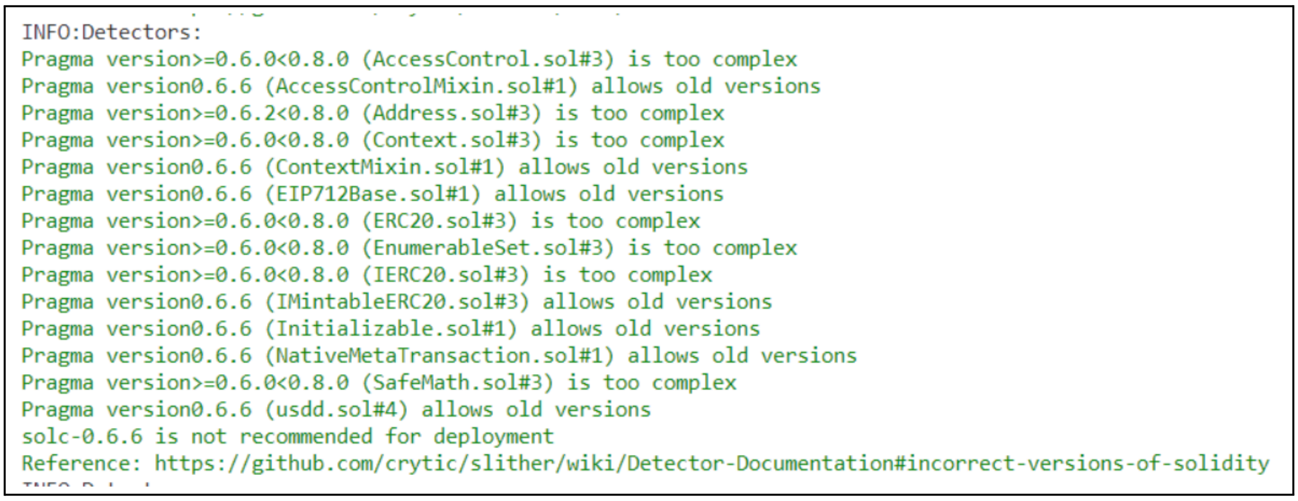


```
INFO:Detectors:
Different versions of Solidity are used:
        - Version used: ['0.6.6', '>=0.6.0<0.8.0', '>=0.6.2<0.8.0']
        - 0.6.6 (AccessControlMixin.sol#1)
        - 0.6.6 (ContextMixin.sol#1)
        - 0.6.6 (EIP712Base.sol#1)
        - 0.6.6 (IMintableERC20.sol#3)
        - 0.6.6 (Initializable.sol#1)
        - 0.6.6 (NativeMetaTransaction.sol#1)
        - 0.6.6 (usdd.sol#4)
        - >=0.6.0<0.8.0 (AccessControl.sol#3)
        - >=0.6.0<0.8.0 (Context.sol#3)
        - >=0.6.0<0.8.0 (ERC20.sol#3)
        - >=0.6.0<0.8.0 (EnumerableSet.sol#3)
        - >=0.6.0<0.8.0 (IERC20.sol#3)
        - >=0.6.0<0.8.0 (SafeMath.sol#3)
        - >=0.6.2<0.8.0 (Address.sol#3)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#different-pragma-directives-are-used
```

Detected different Solidity versions are used.

Resolution: Use one Solidity version.

(2) Use the Latest Solidity Compiler Version for Enhanced Security: **USDD.sol**



```
INFO:Detectors:
Pragma version>=0.6.0<0.8.0 (AccessControl.sol#3) is too complex
Pragma version0.6.6 (AccessControlMixin.sol#1) allows old versions
Pragma version>=0.6.2<0.8.0 (Address.sol#3) is too complex
Pragma version>=0.6.0<0.8.0 (Context.sol#3) is too complex
Pragma version0.6.6 (ContextMixin.sol#1) allows old versions
Pragma version0.6.6 (EIP712Base.sol#1) allows old versions
Pragma version>=0.6.0<0.8.0 (ERC20.sol#3) is too complex
Pragma version>=0.6.0<0.8.0 (EnumerableSet.sol#3) is too complex
Pragma version>=0.6.0<0.8.0 (IERC20.sol#3) is too complex
Pragma version0.6.6 (IMintableERC20.sol#3) allows old versions
Pragma version0.6.6 (Initializable.sol#1) allows old versions
Pragma version0.6.6 (NativeMetaTransaction.sol#1) allows old versions
Pragma version>=0.6.0<0.8.0 (SafeMath.sol#3) is too complex
Pragma version0.6.6 (usdd.sol#4) allows old versions
solc-0.6.6 is not recommended for deployment
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#incorrect-versions-of-solidity
```

Solc frequently releases new compiler versions. Using an old version prevents access to new Solidity security checks. We also recommend avoiding complex pragma statements.

Resolution: Deploy with any of the following Solidity versions:

0.8.18

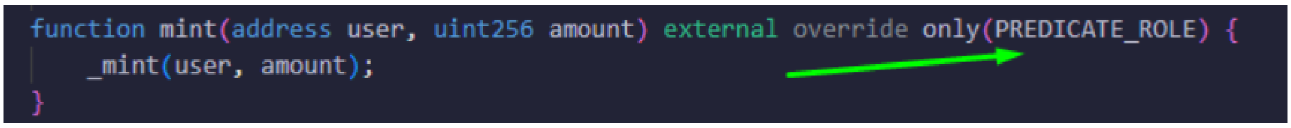
The recommendations take into account:

- Risks related to recent releases
- Risks of complex code generation changes
- Risks of new language features
- Risks of known bugs

Use a simple pragma version that allows any of these versions. Consider using the latest version of Solidity for testing.

(3) Centralized Ownership and Privileges Management:

USDD.sol



```
function mint(address user, uint256 amount) external override only(PREDICATE_ROLE) {
    _mint(user, amount);
}
```

Only PREDICATE_ROLE can mint a usdd token. PREDICATE_ROLE can mint unlimited tokens.

Resolution: Create a contract always 100% Decentralized. Please define the maximum token limit.

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:

AccessControl.sol

- grantRole: Grants `role` to `account` can be set by the admin.
- revokeRole: Revokes `role` from `account` by the admin.
- renounceRole: Renounce Role from `account` by the admin.

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 3 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.

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 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, and 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).

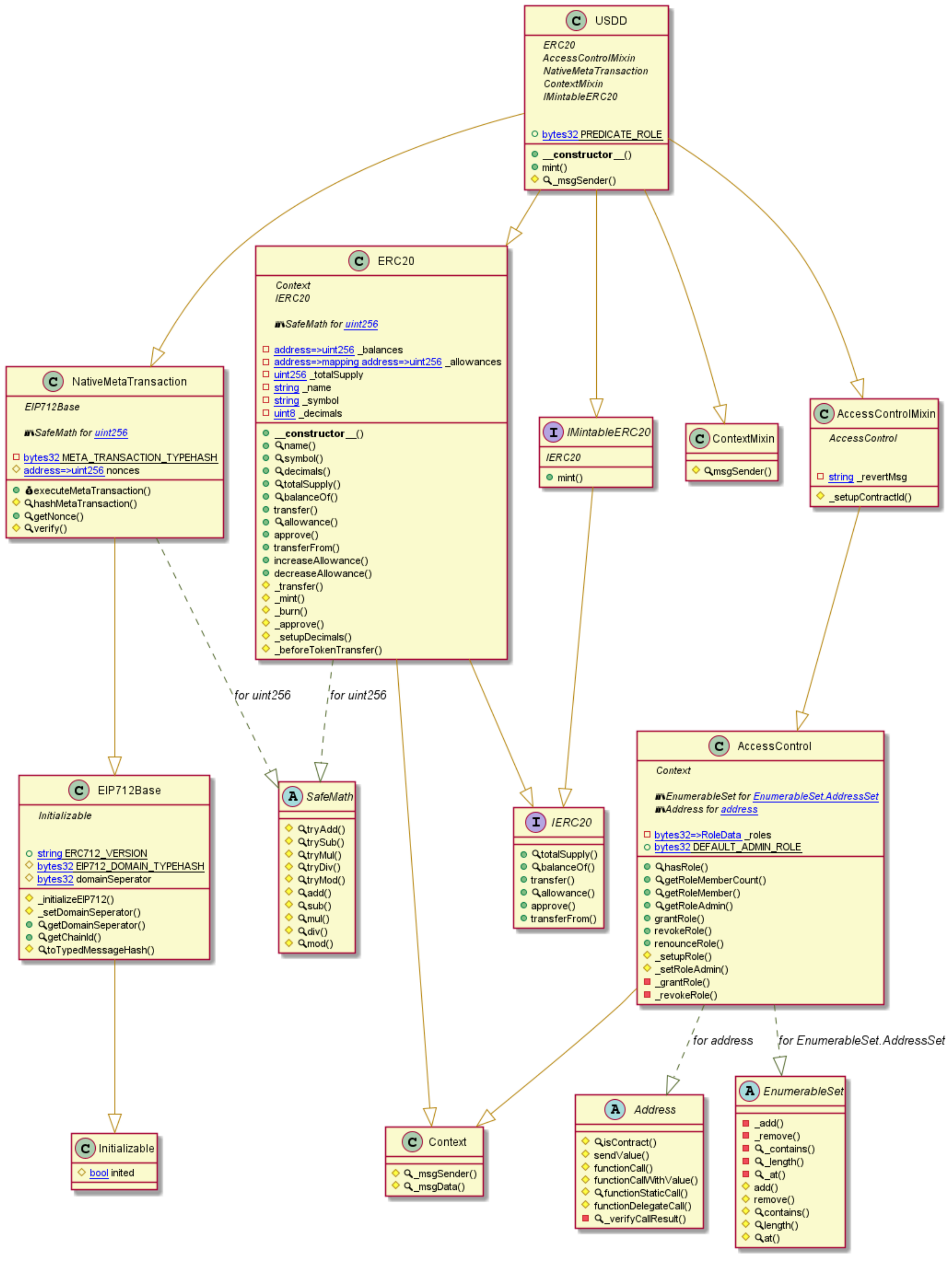
Due to the fact that the total number of test cases is unlimited, the audit makes no statements or warranties on the security of the code. It also cannot be considered as a sufficient assessment regarding the utility and safety of the code, bug-free status or any other statements of the contract. While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

Appendix

Code Flow Diagram - Decentralized USD Token

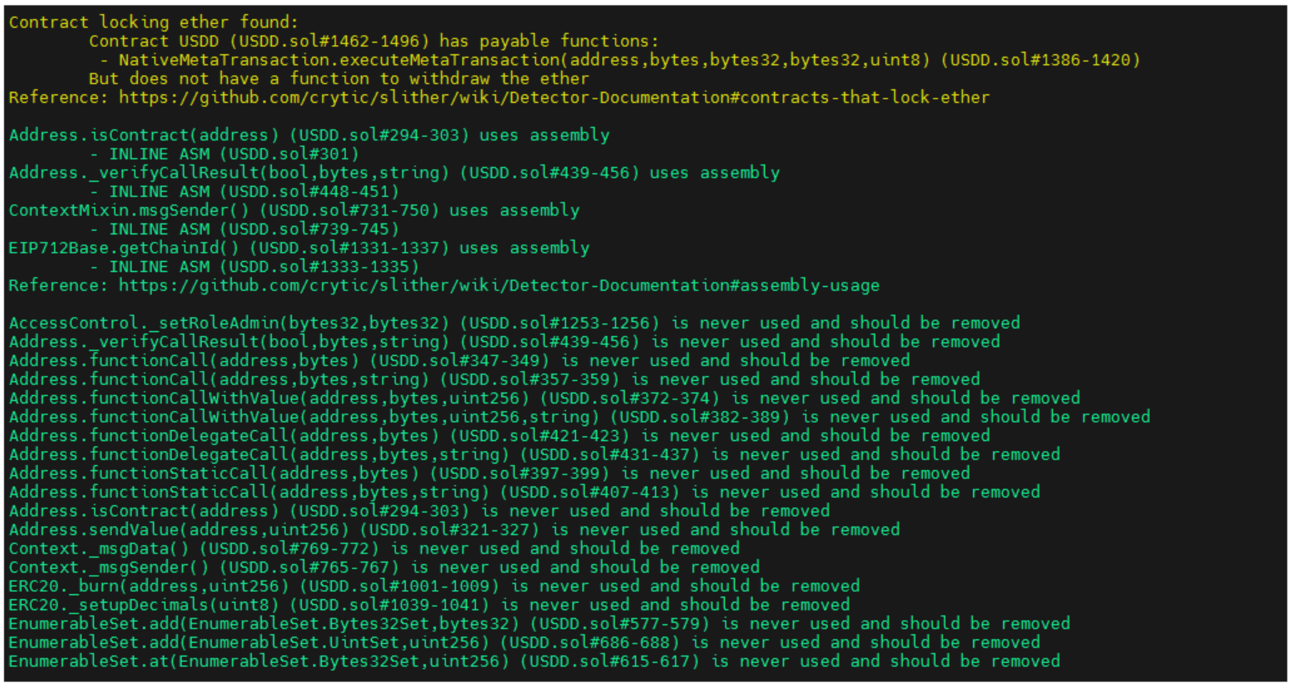


Slither Results Log

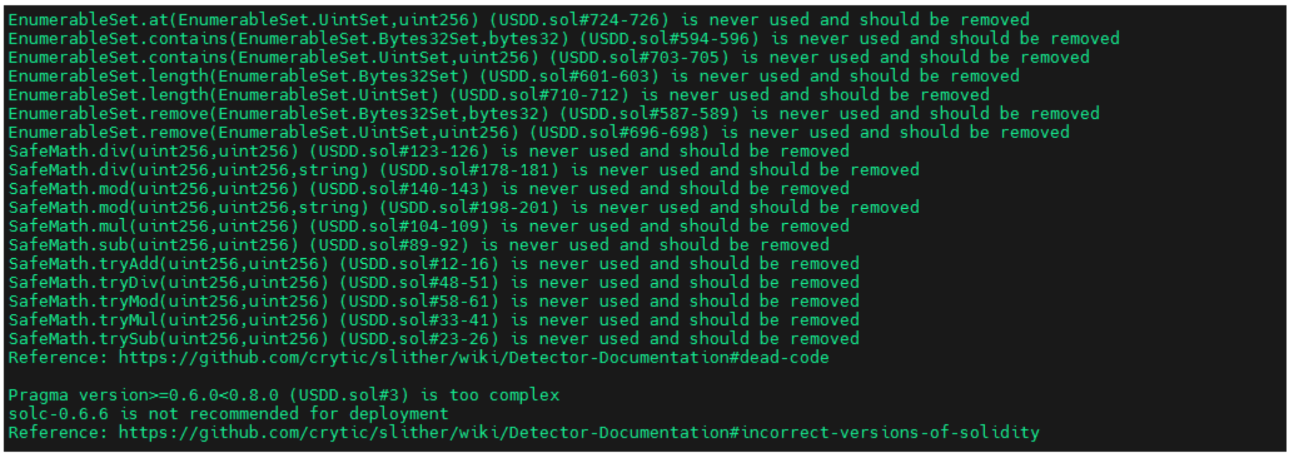
Slither is a Solidity static analysis framework that uses vulnerability detectors, displays contract details, and provides an API for writing custom analyses. It helps developers identify vulnerabilities, improve code comprehension, and prototype custom analyses quickly. The analysis includes a report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

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

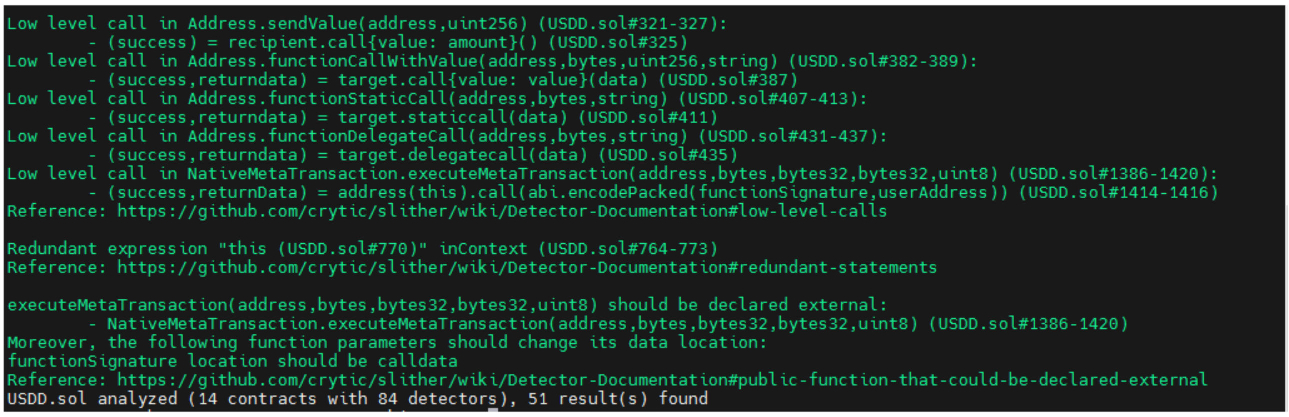
Slither Log >> USDD.sol



```
Contract locking ether found:
        Contract USDD (USDD.sol#1462-1496) has payable functions:
         - NativeMetaTransaction.executeMetaTransaction(address,bytes,bytes32,bytes32,uint8) (USDD.sol#1386-1420)
        But does not have a function to withdraw the ether
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#contracts-that-lock-ether

Address.isContract(address) (USDD.sol#294-303) uses assembly
        - INLINE ASM (USDD.sol#301)
Address._verifyCallResult(bool,bytes,string) (USDD.sol#439-456) uses assembly
        - INLINE ASM (USDD.sol#448-451)
ContextMixin.msgSender() (USDD.sol#731-750) uses assembly
        - INLINE ASM (USDD.sol#739-745)
EIP712Base.getChainId() (USDD.sol#1331-1337) uses assembly
        - INLINE ASM (USDD.sol#1333-1335)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#assembly-usage

AccessControl._setRoleAdmin(bytes32,bytes32) (USDD.sol#1253-1256) is never used and should be removed
Address._verifyCallResult(bool,bytes,string) (USDD.sol#439-456) is never used and should be removed
Address.functionCall(address,bytes) (USDD.sol#347-349) is never used and should be removed
Address.functionCall(address,bytes,string) (USDD.sol#357-359) is never used and should be removed
Address.functionCallWithValue(address,bytes,uint256) (USDD.sol#372-374) is never used and should be removed
Address.functionCallWithValue(address,bytes,uint256,string) (USDD.sol#382-389) is never used and should be removed
Address.functionDelegateCall(address,bytes) (USDD.sol#421-423) is never used and should be removed
Address.functionDelegateCall(address,bytes,string) (USDD.sol#431-437) is never used and should be removed
Address.functionStaticCall(address,bytes) (USDD.sol#397-399) is never used and should be removed
Address.functionStaticCall(address,bytes,string) (USDD.sol#407-413) is never used and should be removed
Address.isContract(address) (USDD.sol#294-303) is never used and should be removed
Address.sendValue(address,uint256) (USDD.sol#321-327) is never used and should be removed
Context._msgData() (USDD.sol#769-772) is never used and should be removed
Context._msgSender() (USDD.sol#765-767) is never used and should be removed
ERC20._burn(address,uint256) (USDD.sol#1001-1009) is never used and should be removed
ERC20._setupDecimals(uint8) (USDD.sol#1039-1041) is never used and should be removed
EnumerableSet.add(EnumerableSet.Bytes32Set,bytes32) (USDD.sol#577-579) is never used and should be removed
EnumerableSet.add(EnumerableSet.UintSet,uint256) (USDD.sol#686-688) is never used and should be removed
EnumerableSet.at(EnumerableSet.Bytes32Set,uint256) (USDD.sol#615-617) is never used and should be removed
```



```
EnumerableSet.at(EnumerableSet.UintSet,uint256) (USDD.sol#724-726) is never used and should be removed
EnumerableSet.contains(EnumerableSet.Bytes32Set,bytes32) (USDD.sol#594-596) is never used and should be removed
EnumerableSet.contains(EnumerableSet.UintSet,uint256) (USDD.sol#703-705) is never used and should be removed
EnumerableSet.length(EnumerableSet.Bytes32Set) (USDD.sol#601-603) is never used and should be removed
EnumerableSet.length(EnumerableSet.UintSet) (USDD.sol#710-712) is never used and should be removed
EnumerableSet.remove(EnumerableSet.Bytes32Set,bytes32) (USDD.sol#587-589) is never used and should be removed
EnumerableSet.remove(EnumerableSet.UintSet,uint256) (USDD.sol#696-698) is never used and should be removed
SafeMath.div(uint256,uint256) (USDD.sol#123-126) is never used and should be removed
SafeMath.div(uint256,uint256,string) (USDD.sol#178-181) is never used and should be removed
SafeMath.mod(uint256,uint256) (USDD.sol#140-143) is never used and should be removed
SafeMath.mod(uint256,uint256,string) (USDD.sol#198-201) is never used and should be removed
SafeMath.mul(uint256,uint256) (USDD.sol#104-109) is never used and should be removed
SafeMath.sub(uint256,uint256) (USDD.sol#89-92) is never used and should be removed
SafeMath.tryAdd(uint256,uint256) (USDD.sol#12-16) is never used and should be removed
SafeMath.tryDiv(uint256,uint256) (USDD.sol#48-51) is never used and should be removed
SafeMath.tryMod(uint256,uint256) (USDD.sol#58-61) is never used and should be removed
SafeMath.tryMul(uint256,uint256) (USDD.sol#33-41) is never used and should be removed
SafeMath.trySub(uint256,uint256) (USDD.sol#23-26) is never used and should be removed
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#dead-code

Pragma version>=0.6.0<0.8.0 (USDD.sol#3) is too complex
solc-0.6.6 is not recommended for deployment
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#incorrect-versions-of-solidity
```

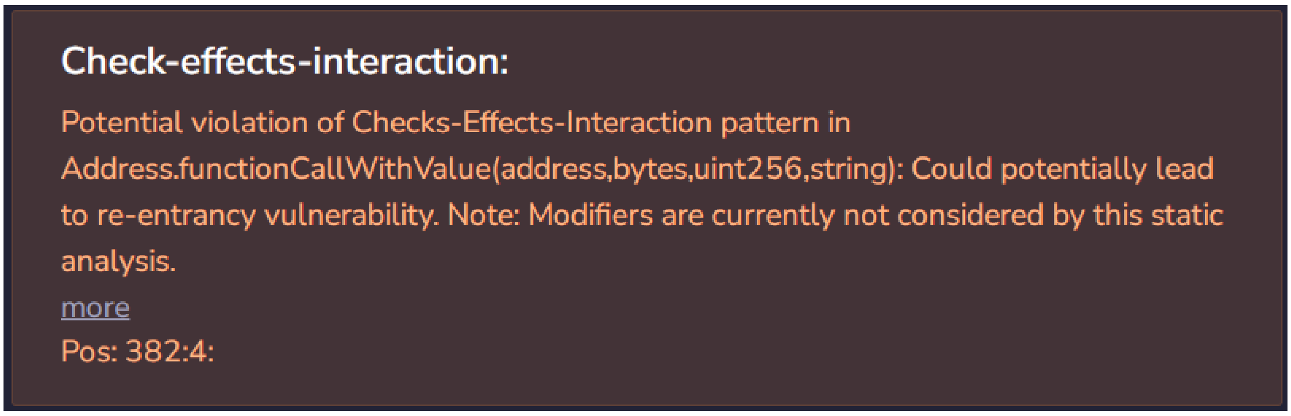


```
Low level call in Address.sendValue(address,uint256) (USDD.sol#321-327):
        - (success) = recipient.call{value: amount}() (USDD.sol#325)
Low level call in Address.functionCallWithValue(address,bytes,uint256,string) (USDD.sol#382-389):
        - (success,returndata) = target.call{value: value}(data) (USDD.sol#387)
Low level call in Address.functionStaticCall(address,bytes,string) (USDD.sol#407-413):
        - (success,returndata) = target.staticcall(data) (USDD.sol#411)
Low level call in Address.functionDelegateCall(address,bytes,string) (USDD.sol#431-437):
        - (success,returndata) = target.delegatecall(data) (USDD.sol#435)
Low level call in NativeMetaTransaction.executeMetaTransaction(address,bytes,bytes32,bytes32,uint8) (USDD.sol#1386-1420):
        - (success,returnData) = address(this).call(abi.encodePacked(functionSignature,userAddress)) (USDD.sol#1414-1416)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#low-level-calls

Redundant expression "this (USDD.sol#770)" inContext (USDD.sol#764-773)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#redundant-statements

executeMetaTransaction(address,bytes,bytes32,bytes32,uint8) should be declared external:
        - NativeMetaTransaction.executeMetaTransaction(address,bytes,bytes32,bytes32,uint8) (USDD.sol#1386-1420)
Moreover, the following function parameters should change its data location:
functionSignature location should be calldata
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#public-function-that-could-be-declared-external
USDD.sol analyzed (14 contracts with 84 detectors), 51 result(s) found
```

Solidity Static Analysis

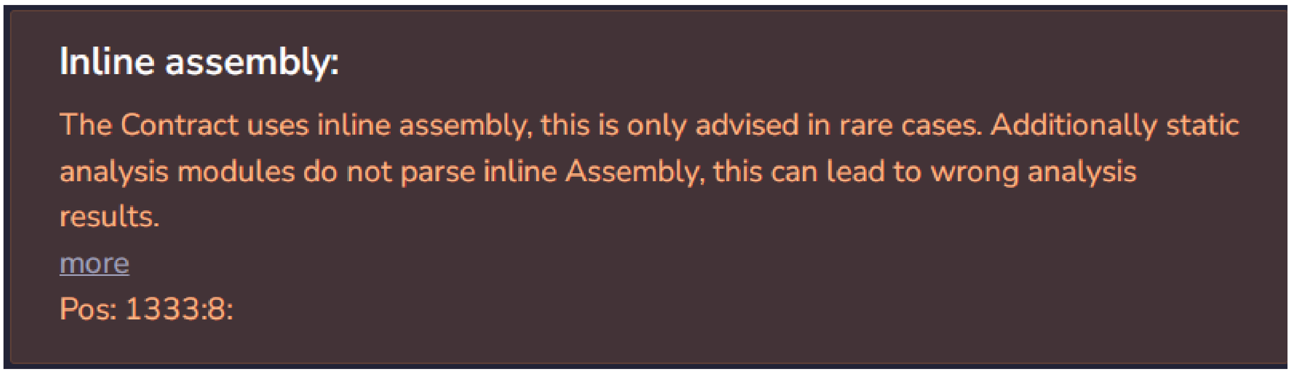
Static code analysis is used to identify many common coding problems before a program is released. It involves examining the code manually or using tools to automate the process. Static code analysis tools can automatically scan the code without executing it.

USDD.sol



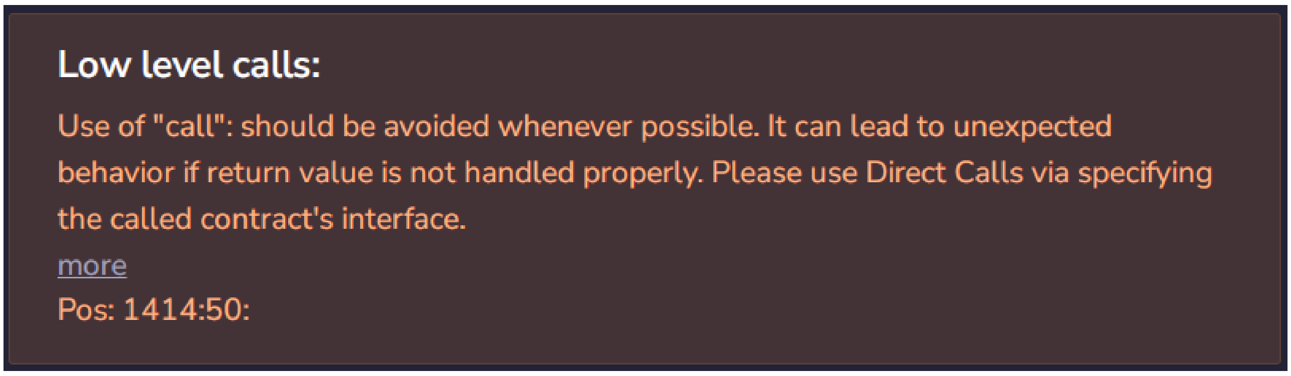
Check-effects-interaction:

Potential violation of Checks-Effects-Interaction pattern in Address.functionCallWithValue(address,bytes,uint256,string): Could potentially lead to re-entrancy vulnerability. Note: Modifiers are currently not considered by this static analysis.
more
Pos: 382:4:



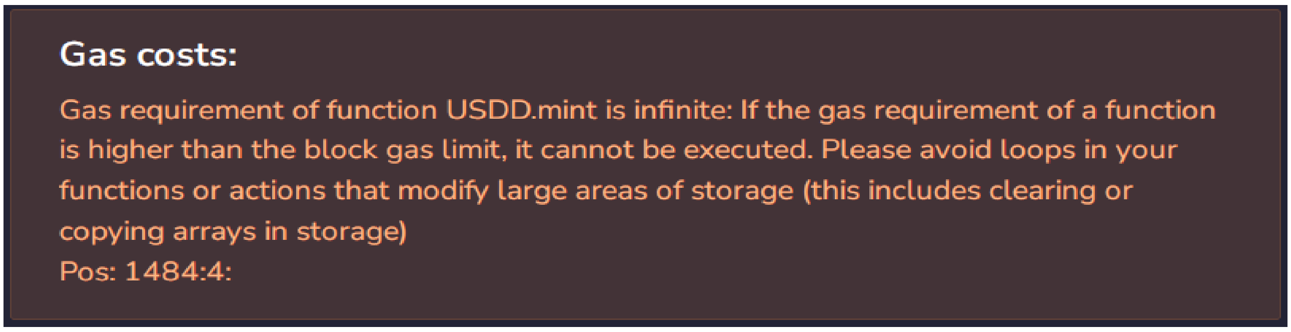
Inline assembly:

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: 1333:8:



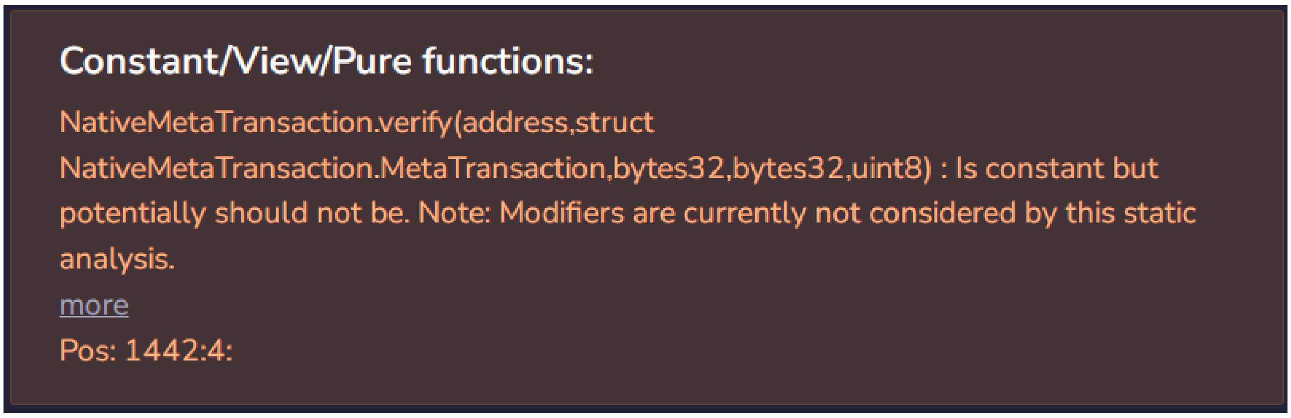
Low level calls:

Use of "call": should be avoided whenever possible. It can lead to unexpected behavior if return value is not handled properly. Please use Direct Calls via specifying the called contract's interface.
more
Pos: 1414:50:



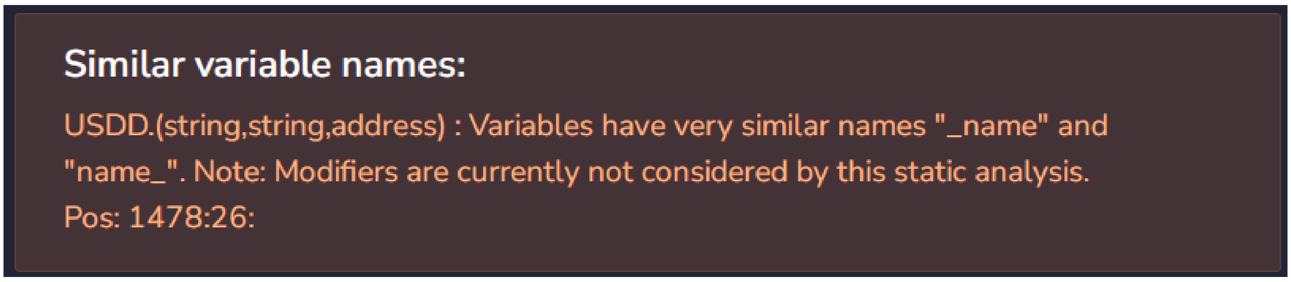
Gas costs:

Gas requirement of function USDD.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: 1484:4:



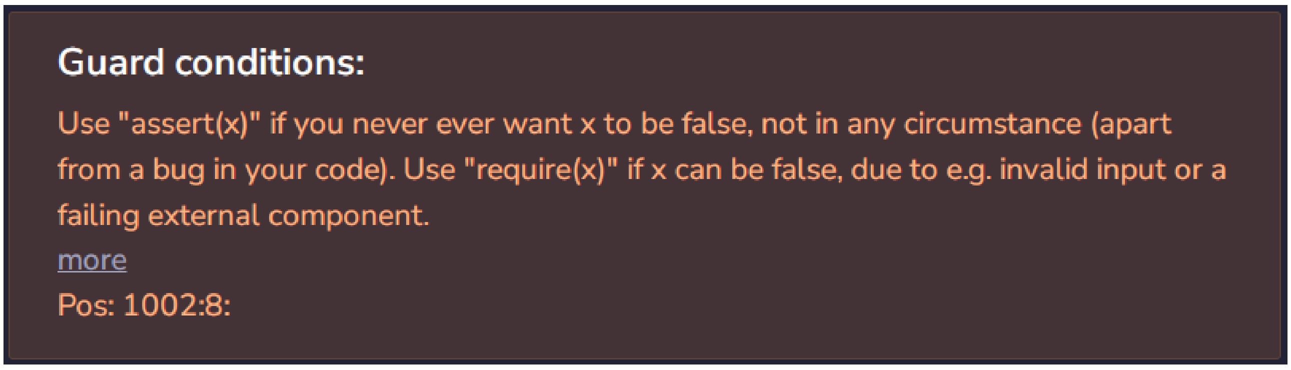
Constant/View/Pure functions:

NativeMetaTransaction.verify(address,struct NativeMetaTransaction.MetaTransaction,bytes32,bytes32,uint8) : Is constant but potentially should not be. Note: Modifiers are currently not considered by this static analysis.
more
Pos: 1442:4:



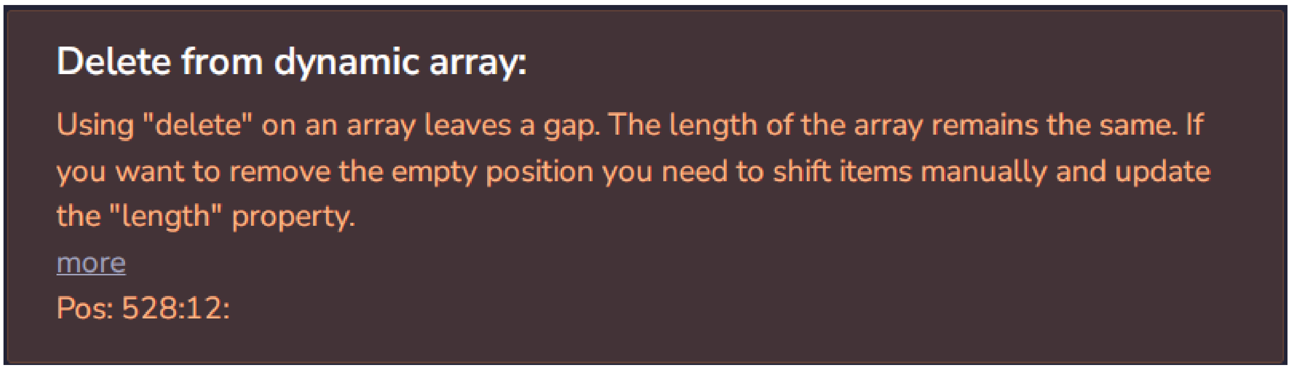
Similar variable names:

USDD.(string,string,address) : Variables have very similar names "_name" and "name_". Note: Modifiers are currently not considered by this static analysis.
Pos: 1478:26:



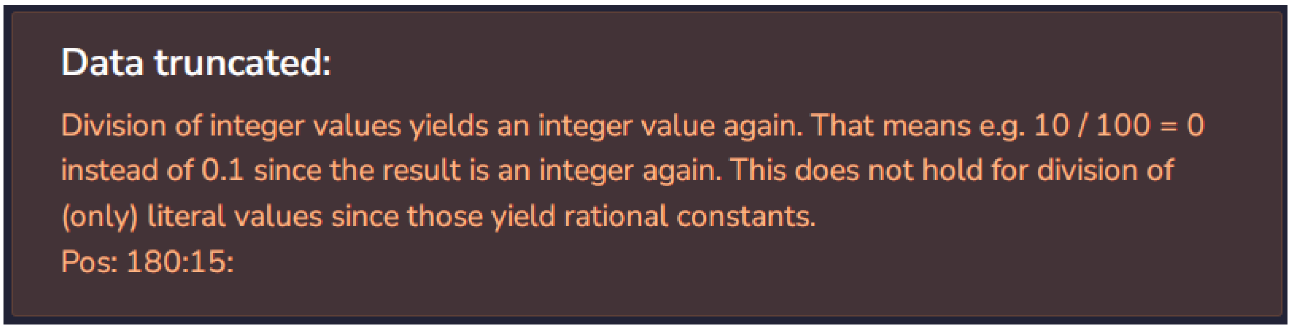
Guard conditions:

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: 1002:8:



Delete from dynamic array:

Using "delete" on an array leaves a gap. The length of the array remains the same. If you want to remove the empty position you need to shift items manually and update the "length" property.
more
Pos: 528:12:



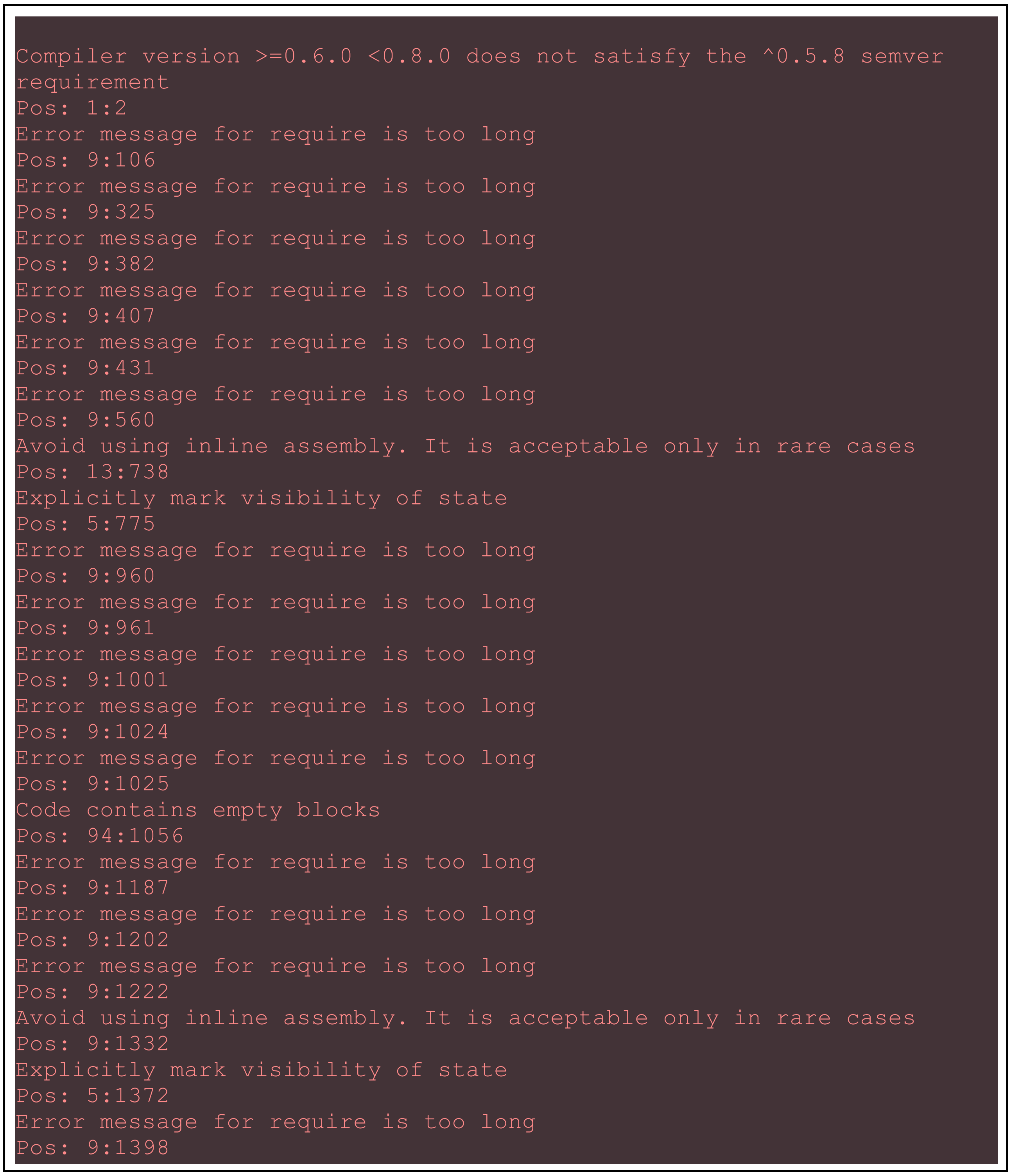
Data truncated:

Division of integer values yields an integer value again. That means e.g. 10 / 100 = 0 instead of 0.1 since the result is an integer again. This does not hold for division of (only) literal values since those yield rational constants.
Pos: 180:15:

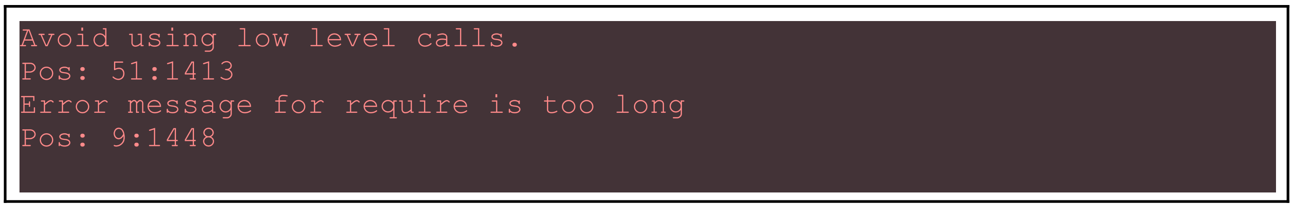
Solhint Linter

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.

USDD.sol



```
Compiler version >=0.6.0 <0.8.0 does not satisfy the ^0.5.8 semver
requirement
Pos: 1:2
Error message for require is too long
Pos: 9:106
Error message for require is too long
Pos: 9:325
Error message for require is too long
Pos: 9:382
Error message for require is too long
Pos: 9:407
Error message for require is too long
Pos: 9:431
Error message for require is too long
Pos: 9:560
Avoid using inline assembly. It is acceptable only in rare cases
Pos: 13:738
Explicitly mark visibility of state
Pos: 5:775
Error message for require is too long
Pos: 9:960
Error message for require is too long
Pos: 9:961
Error message for require is too long
Pos: 9:1001
Error message for require is too long
Pos: 9:1024
Error message for require is too long
Pos: 9:1025
Code contains empty blocks
Pos: 94:1056
Error message for require is too long
Pos: 9:1187
Error message for require is too long
Pos: 9:1202
Error message for require is too long
Pos: 9:1222
Avoid using inline assembly. It is acceptable only in rare cases
Pos: 9:1332
Explicitly mark visibility of state
Pos: 5:1372
Error message for require is too long
Pos: 9:1398
```



```
Avoid using low level calls.
Pos: 51:1413
Error message for require is too long
Pos: 9:1448
```

Software analysis result:

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