EtherAuthority

www.EtherAuthority.io
audit@etherauthority.io

SMART CONTRACT

Security Audit Report

Project:	USD Coin (PoS) (USDC.e)
Website:	portal.polygon.technology
Platform:	Polygon
Language:	Solidity
Date:	April 8th, 2025

Table of Contents

Introduction

As part of EtherAuthority's community smart contract audit initiatives, the smart contract of the USDC.e Token from portal.polygon.technology 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 8th, 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

The UChildAdministrableERC20 contract is an advanced child-chain compatible ERC20 token implementation. It extends multiple contracts to incorporate key administrative and operational features. Here's a breakdown of the **details and functionality**:

Inheritance Structure

UChildAdministrableERC20 inherits from:

- UChildERC20 – Base child token implementation compatible with the Polygon POS bridge.
- Blacklistable – Provides address blacklisting capability.
- Pausable – Allows pausing and unpausing of token operations.
- Rescuable – Enables rescuing stuck ERC20 tokens by admins.

Audit scope

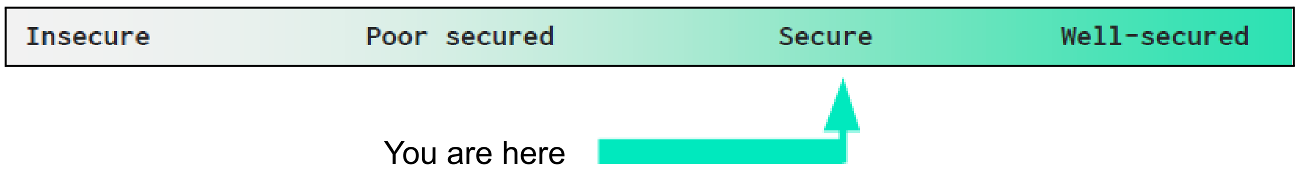
Name	Code Review and Security Analysis Report for USD Coin (PoS) (USDC.e) Token Smart Contracts
Platform	**Polygon**
File 1	UChildAdministrableERC20.sol
File 1 Smart Contract	0xdd9185db084f5c4fff3b4f70e7ba62123b812226
File 2	UChildERC20Proxy.sol
File 2 Smart Contract	0x2791bca1f2de4661ed88a30c99a7a9449aa84174
Audit Date	April 8th, 2025

Claimed Smart Contract Features

Claimed Feature Detail	Our Observation
Key Features: The UChildAdministrableERC20 contract is designed as a feature-rich, multi-layered ERC20 implementation. It combines: ● Polygon bridging functionality ● Upgradeable and meta-transaction support ● Advanced gasless token permissions (permit & authorizations) ● Robust administrative controls (blacklist, pause, rescue) ● Granular role-based access control This makes it well-suited for applications where regulatory requirements, user-friendly experiences (via gasless transactions), and the ability to respond swiftly to emerging threats are paramount. Whether you are deploying on Polygon or integrating advanced token functionalities in a broader ecosystem, the architectural design of this contract helps ensure a balance between flexibility, security, and administrative control.	**YES, This is valid.**

Audit Summary

According to the standard audit assessment, the Customer's solidity-based smart contract is **"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, 1 low, and 3 very low-level issues.

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	Moderated
	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.e 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.e 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.e 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

UChildAdministrableERC20.sol : Functions

Sl.	Functions	Type	Observation	Conclusion
1	_msgSender	internal	Passed	No Issue
2	withdraw	external	notBlacklisted	No Issue
3	transfer	external	notBlacklisted	No Issue
4	approve	external	notBlacklisted	No Issue
5	transferFrom	external	notBlacklisted	No Issue
6	increaseAllowance	external	notBlacklisted	No Issue
7	decreaseAllowance	external	notBlacklisted	No Issue
8	permit	external	notBlacklisted	No Issue
9	transferWithAuthorization	external	notBlacklisted	No Issue
10	approveWithAuthorization	external	notBlacklisted	No Issue
11	increaseAllowanceWithAuthorization	external	notBlacklisted	No Issue
12	withdrawWithAuthorization	external	notBlacklisted	No Issue
13	decreaseAllowanceWithAuthorization	external	notBlacklisted	No Issue
14	cancelAuthorization	external	whenNotPaused	No Issue
15	initialize	external	initializer	No Issue
16	_msgSender	internal	Passed	No Issue
17	updateMetadata	external	DEFAULT_ADMIN_ROLE	No Issue
18	deposit	external	Unchecked `msg.data` in `deposit()` Function,No Cap on Token Supply	Refer Audit Findings
19	withdraw	external	Passed	No Issue
20	permit	external	Passed	No Issue
21	transferWithAuthorization	external	Passed	No Issue
22	approveWithAuthorization	external	Passed	No Issue
23	increaseAllowanceWithAuthorization	external	Passed	No Issue
24	decreaseAllowanceWithAuthorization	external	Passed	No Issue
25	withdrawWithAuthorization	external	Passed	No Issue
26	withdrawWithAuthorization	external	Passed	No Issue
27	notBlacklisted	modifier	Passed	No Issue
28	blacklisters	external	Passed	No Issue
29	isBlacklisted	external	Passed	No Issue
30	blacklist	external	Lack of Event Emissions for Critical Functions	Refer Audit Findings

31	unBlacklist	external	Lack of Event Emissions for Critical Functions	Refer Audit Findings
32	whenNotPaused	modifier	Passed	No Issue
33	pausers	external	Passed	No Issue
34	paused	external	Passed	No Issue
35	pause	external	Lack of Event Emissions for Critical Functions	Refer Audit Findings
36	unpause	external	Lack of Event Emissions for Critical Functions	Refer Audit Findings
37	rescuers	external	Passed	No Issue
38	rescueERC20	external	Lack of Event Emissions for Critical Functions, \`rescueERC20\` Allows Arbitrary Withdrawals	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

(1) Lack of Event Emissions for Critical Functions:

Functions like `blacklist`, `unblacklist`, `pause`, `unpause`, and `rescueERC20` may not emit detailed custom events.

Resolution: Emit events such as `Blacklisted(address)` and `UnBlacklisted(address)` when the blacklist status changes.

Very Low / Informational / Best practices:

(1) Unchecked `msg.data` in `deposit()` Function:

If the `deposit` function decodes `amount` from `bytes calldata depositData` without proper validation, it could lead to malformed input, unintended minting, or runtime errors.

Resolution: Ensure validation of `depositData`:

```
require(depositData.length == 32, "Invalid deposit data");

uint256 amount = abi.decode(depositData, (uint256));
```

(2) `rescueERC20` Allows Arbitrary Withdrawals:

If the contract holds user-deposited tokens or interacts with other contracts, allowing unrestricted token rescue could lead to fund loss.

Resolution: Restrict `rescueERC20`:

- Only allow rescue of tokens *not* native to the contract.
- Add a whitelist of rescable tokens.
- Log all rescues with detailed events.

(3) No Cap on Token Supply:

The `mint()` in `deposit()` has no upper limit, which could be abused in case of a bug in the root chain's bridge or an unauthorized depositor.

Resolution: Add an optional `maxSupply`:

- require(totalSupply() + amount <= maxSupply, "Exceeds max supply");

Centralization Risk

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

UChildERC20.sol

- updateMetadata: The admin can update the metadata.

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 low and 3 informational issues 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).

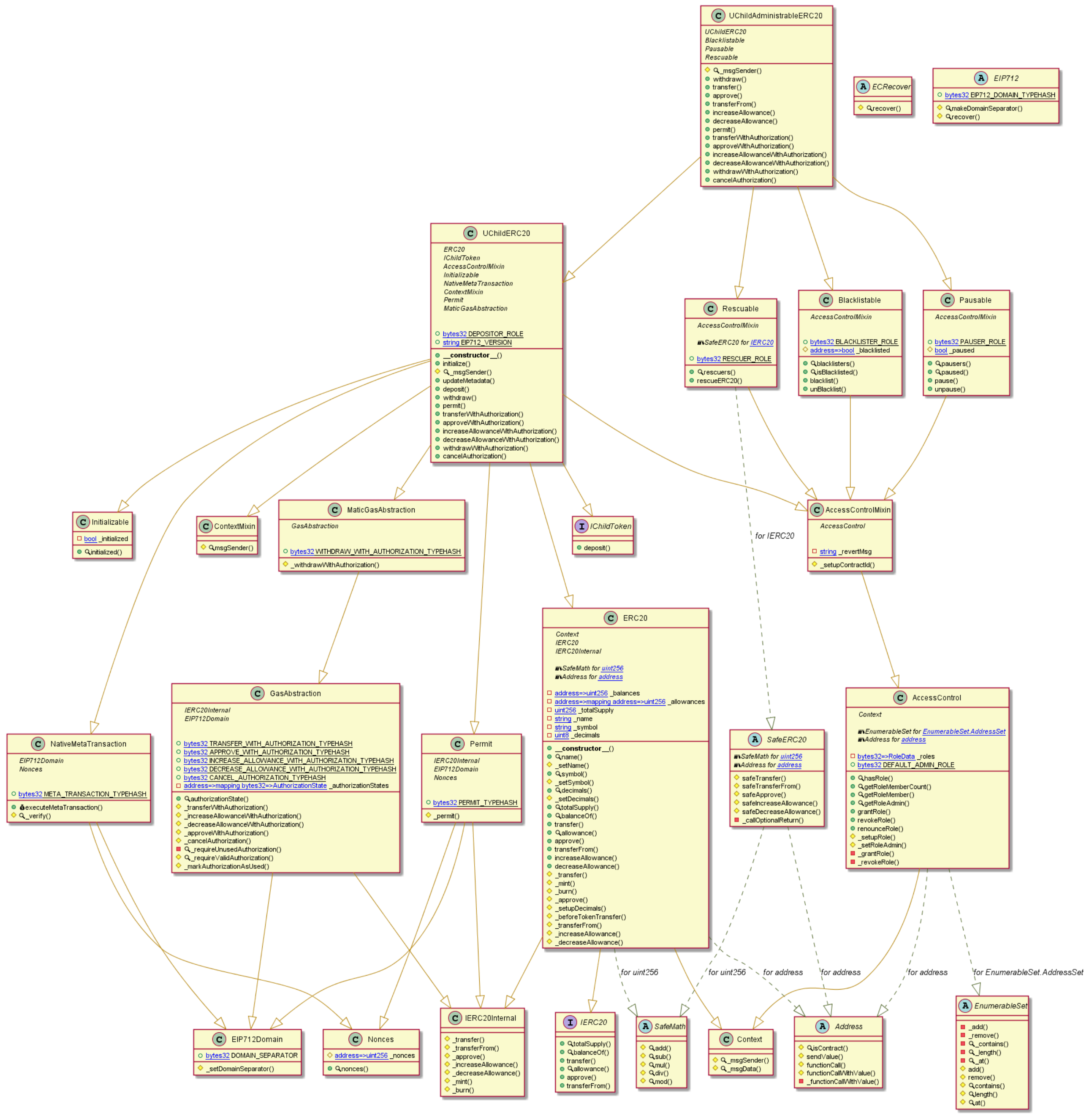
Because the total number of test cases is unlimited, the audit makes no statements or warranties on the security of the code. It also cannot be considered as a sufficient assessment regarding the utility and safety of the code, bug-free status, or any other statements of the contract. While we have done 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.

Appendix

Code Flow Diagram - USD Coin (PoS) (USDC.e)) Token



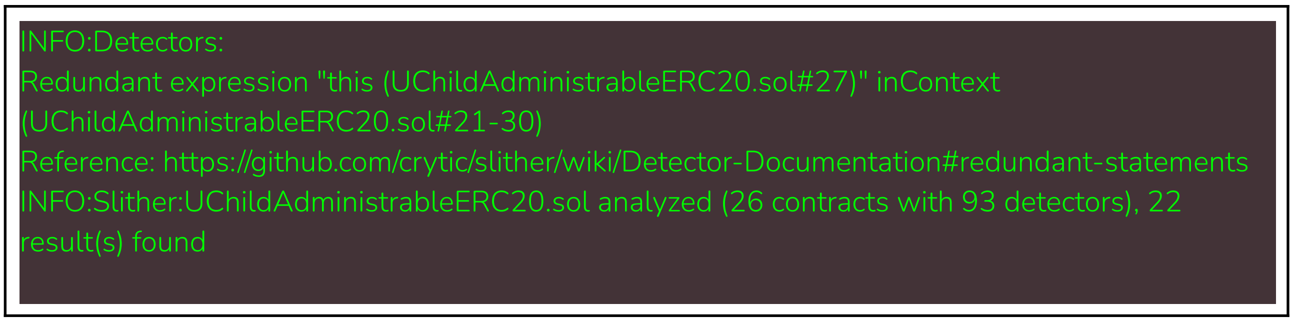
Slither Results Log

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 >> UChildAdministrableERC20.sol

```
INFO:Detectors:
Permit._permit(address,address,uint256,uint256,uint8,bytes32,bytes32)
(UChildAdministrableERC20.sol#1843-1869) uses timestamp for comparisons
	Dangerous comparisons:
	- require(bool,string)(deadline >= now,Permit: permit is expired)
(UChildAdministrableERC20.sol#1853)
GasAbstraction._requireValidAuthorization(address,bytes32,uint256,uint256)
(UChildAdministrableERC20.sol#2178-2190) uses timestamp for comparisons
	Dangerous comparisons:
	- require(bool,string)(now > validAfter,GasAbstraction: authorization is not yet valid)
(UChildAdministrableERC20.sol#2184-2187)
	- require(bool,string)(now < validBefore,GasAbstraction: authorization is expired)
(UChildAdministrableERC20.sol#2188)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#block-timestamp
INFO:Detectors:
2 different versions of Solidity are used:
	- Version constraint ^0.6.12 is used by:
		-^0.6.12 (UChildAdministrableERC20.sol#9)
Reference:
https://github.com/crytic/slither/wiki/Detector-Documentation#different-pragma-directives-are-us
ed
INFO:Detectors:
UChildERC20._msgSender() (UChildAdministrableERC20.sol#2284-2292) is never used and
should be removed
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#dead-code
INFO:Detectors:
Variable EIP712Domain.DOMAIN_SEPARATOR (UChildAdministrableERC20.sol#1672) is not in
mixedCase
Reference:
https://github.com/crytic/slither/wiki/Detector-Documentation#conformance-to-solidity-naming-c
onventions
```

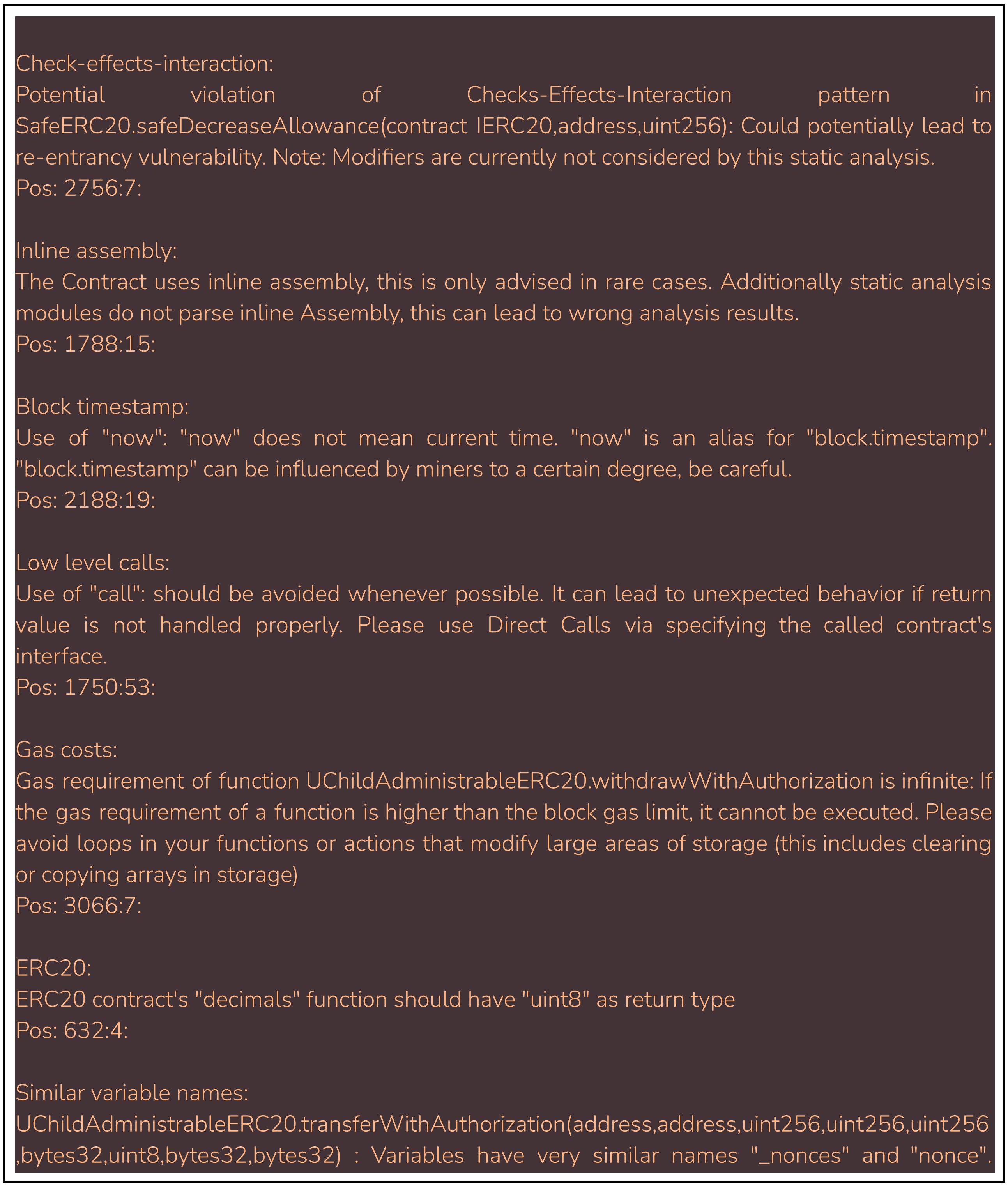


```
INFO:Detectors:
Redundant expression "this (UChildAdministrableERC20.sol#27)" inContext
(UChildAdministrableERC20.sol#21-30)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#redundant-statements
INFO:Slither:UChildAdministrableERC20.sol analyzed (26 contracts with 93 detectors), 22
result(s) found
```

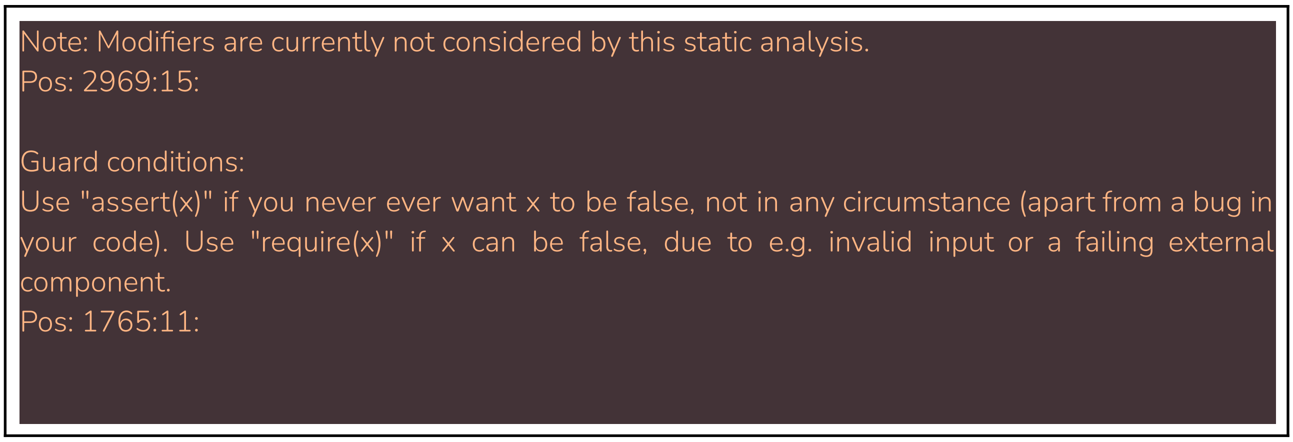
Solidity Static Analysis

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.

UChildAdministrableERC20.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: 2756:7:

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: 1788:15:

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: 2188:19:

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.
Pos: 1750:53:

Gas costs:
Gas requirement of function UChildAdministrableERC20.withdrawWithAuthorization 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: 3066:7:

ERC20:
ERC20 contract's "decimals" function should have "uint8" as return type
Pos: 632:4:

Similar variable names:
UChildAdministrableERC20.transferWithAuthorization(address,address,uint256,uint256,uint256
,bytes32,uint8,bytes32,bytes32) : Variables have very similar names "_nonces" and "nonce".
```

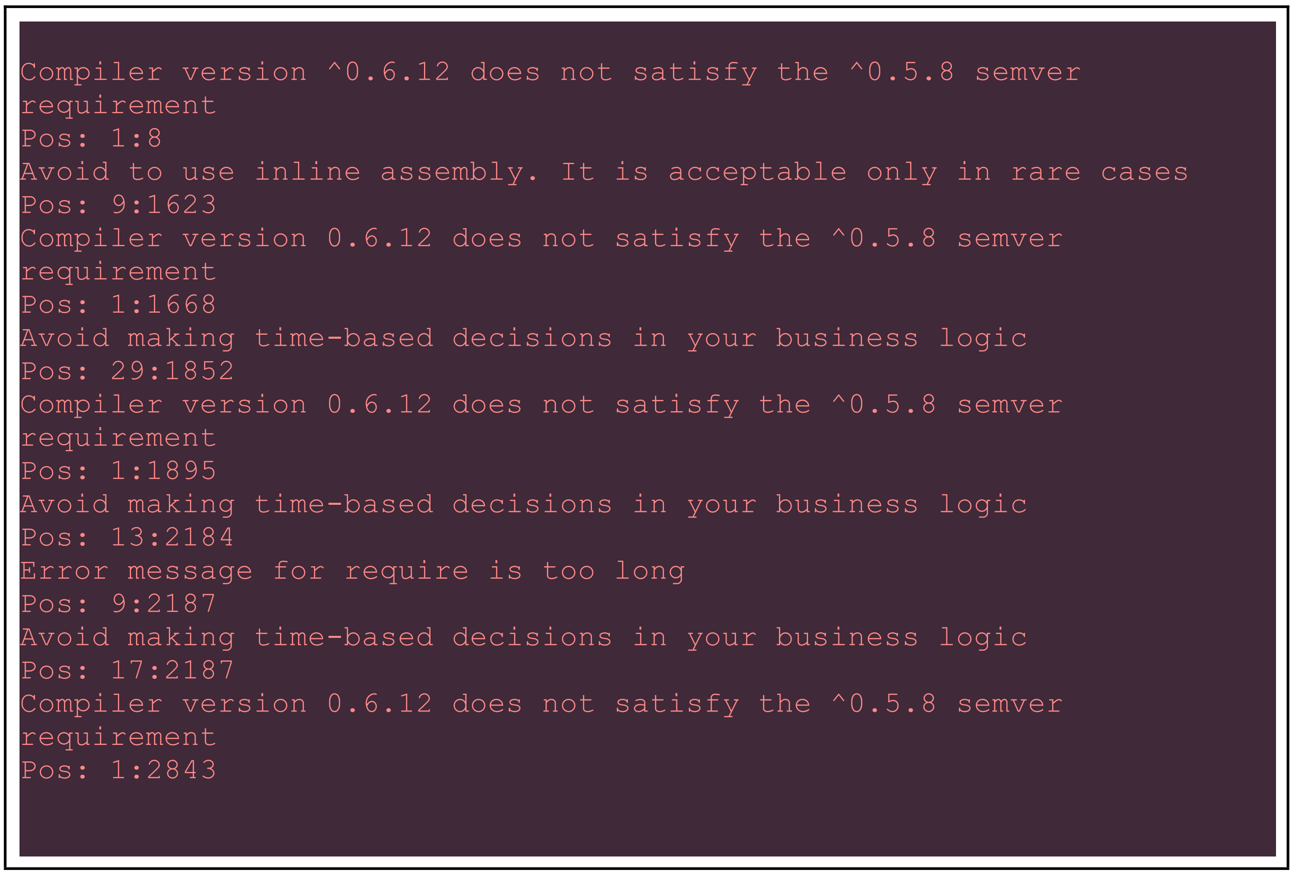


```
Note: Modifiers are currently not considered by this static analysis.
Pos: 2969:15:

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: 1765:11:
```

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.

UChildAdministrableERC20.sol



```
Compiler version ^0.6.12 does not satisfy the ^0.5.8 semver
requirement
Pos: 1:8
Avoid to use inline assembly. It is acceptable only in rare cases
Pos: 9:1623
Compiler version 0.6.12 does not satisfy the ^0.5.8 semver
requirement
Pos: 1:1668
Avoid making time-based decisions in your business logic
Pos: 29:1852
Compiler version 0.6.12 does not satisfy the ^0.5.8 semver
requirement
Pos: 1:1895
Avoid making time-based decisions in your business logic
Pos: 13:2184
Error message for require is too long
Pos: 9:2187
Avoid making time-based decisions in your business logic
Pos: 17:2187
Compiler version 0.6.12 does not satisfy the ^0.5.8 semver
requirement
Pos: 1:2843
```

Software analysis result:

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