Ether Authority

www.EtherAuthority.io
audit@etherauthority.io

SMART CONTRACT

Security Audit Report

Project: Wrapped Ether
Website: weth.io
Platform: Polygon
Language: Solidity
Date: April 4th, 2025

Table of Contents

Introduction 4

Project Background 4

Audit Scope 5

Claimed Smart Contract Features 6

Audit Summary 8

Technical Quick Stats 9

Code Quality 10

Documentation 10

Use of Dependencies 10

AS-IS overview 11

Severity Definitions 12

Audit Findings 13

Conclusion 16

Our Methodology 17

Disclaimers 19

Appendix

- Code Flow Diagram 20
- Slither Results Log 21
- Solidity static analysis 23
- Solhint Linter 24

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 Wrapped Ether Token from weth.io 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 4th, 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

This Solidity codebase implements a customizable and upgradeable **ERC20 token** with support for **access control**, **meta-transactions**, and **modular extensibility**. The design is tailored for use cases such as tokenized platforms, dApps on Polygon, or smart wallets that require gasless interactions and permissioned roles.

Core Features

1. ERC20 Token Standard

- Implements full ERC20 functionality: transfer, approve, transferFrom, etc.
- Uses SafeMath to prevent overflows/underflows.
- Internal _transfer, _mint, and _burn functions give flexibility to extended contracts.

2. Access Control

- Uses AccessControl with EnumerableSet to manage roles (e.g., MINTER, ADMIN).
- Role-based permission system replaces Ownable for more granular security.

3. Initializable (Proxy Support)

- Uses Initializable from OpenZeppelin, making the contract **upgradeable** and suitable for proxy deployments via tools like OpenZeppelin Upgrades or Hardhat.

4. EIP-712 Meta-Transaction Support

- Implements EIP712Base and NativeMetaTransaction for gasless interactions.
- Allows users to sign transactions off-chain while relayers execute them on-chain.
- Enhances UX for dApps by avoiding direct ETH payments from users.

5. Extensibility & Modularity

- The design allows contracts to inherit and extend functionality cleanly.
- AccessControlMixin, Context, and interfaces are used to build a layered system.

Audit scope

Name	**Code Review and Security Analysis Report for Wrapped Ether Token Smart Contract**
Platform	**Polygon**
File	MaticWETH.sol
Smart Contract Code	0x7ceb23fd6bc0add59e62ac25578270cff1b9f619
Audit Date	April 4th, 2025

Claimed Smart Contract Features

Claimed Feature Detail	Our Observation
Tokenomics: • Name: Wrapped Ether • Symbol: WETH • Decimals: 18	**YES, This is valid.**
Key Features: **1.ERC20 Token Implementation** • Fully compliant with the ERC20 standard. • Supports transfer, approve, transferFrom, balanceOf, and totalSupply. • Safe arithmetic via OpenZeppelin's SafeMath. **2. Upgradeable via Initializer** • Uses OpenZeppelin's Initializable pattern to allow proxy-based upgrades. • State variables are initialized once via the initialize function. • Ensures safe deployment using upgradeable smart contract patterns. **3. Access Control with Roles** • Replaces Ownable with flexible AccessControlMixin. • Uses MINTER_ROLE, DEFAULT_ADMIN_ROLE, and any custom roles you define. • Powered by EnumerableSet for efficient role enumeration.	**YES, This is valid.**

4. Meta-Transaction Support (EIP-712) ● Integrates NativeMetaTransaction and EIP712Base. ● Users can interact **gaslessly** using signed messages. ● Ideal for dApps with Web3 wallets, mobile apps, and relayer-based systems. **5. Context and Execution Handling** ● Uses a custom ContextMixin to abstract msgSender() for meta-tx compatibility. ● Ensures the right message sender is identified, whether via proxy or direct call. **6. Polygon Compatibility** ● Contract designed with Polygon dApps in mind. ● Optimized for gas efficiency and meta-transaction interactions, which are common on Polygon.	

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, 2 low, and 4 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 is too old	Moderated
	Integer overflow/underflow	Passed
	Function input parameters lack of check	Passed
	Function input parameters check bypass	Passed
	Function access control lacks management	Moderated
	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 Wrapped Ether 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 Wrapped Ether 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 Wrapped Ether 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

MaticWETH.sol : Functions

Sl.	Functions	Type	Observation	Conclusion
1	constructor	write	Passed	No Issue
2	_msgSender	internal	Passed	No Issue
3	deposit	external	No Return on `deposit`	Refer Audit Findings
4	withdraw	external	No Event on Withdraw	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) No Event on Withdraw:

The `withdraw()` function burns tokens but emits no event for the withdrawal, which may reduce off-chain observability.

Resolution: Emit a custom `Withdraw(address user, uint256 amount)` event in the `withdraw()` function.

(2) Lack of Access Control on `executeMetaTransaction`:

Any address can call `executeMetaTransaction`, which is expected, but it means any relayer can spam your contract with signed transactions (including expired/replayed ones if nonce isn't checked properly).

Resolution: Ensure the off-chain signature process is secure and user wallet libraries validate nonces. Optionally, allow whitelisted relayers only.

Very Low / Informational / Best practices:

(1) No Return on `deposit():

Although not necessary, having a return value or at least emitting a `Deposit` event is best practice.

Resolution: Emit a `Deposit(address user, uint256 amount)` event inside the `deposit()` function.

(2) Solidity Version is 0.6.x (Outdated):

The contracts are written in Solidity `0.6.x`, which lacks some safety and gas optimizations of `0.8.x`.

Resolution: Upgrade to Solidity `^0.8.0` if possible to benefit from overflow checks and better memory handling.

(3) Use of `SafeMath` in Solidity >=0.8.0 is Unnecessary:

If you upgrade to Solidity 0.8+, overflow checks are built in, making `SafeMath` redundant.

Resolution: If you upgrade to Solidity 0.8+, overflow checks are built in, making `SafeMath` redundant.

(4) Missing `receive()` or `fallback()` Function:

The contract accepts `msg.value` in `executeMetaTransaction`, but has no `receive()` or `fallback()` defined. If someone sends ETH accidentally or via `transfer`, it will be rejected.

Resolution: Implement a `receive()` or `fallback()` if the contract should be able to accept ETH safely.

Centralization Risk

The WETH 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 2 low and 4 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 standard audit procedure scope, is **"Secured".**

Email: audit@EtherAuthority.io

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 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).

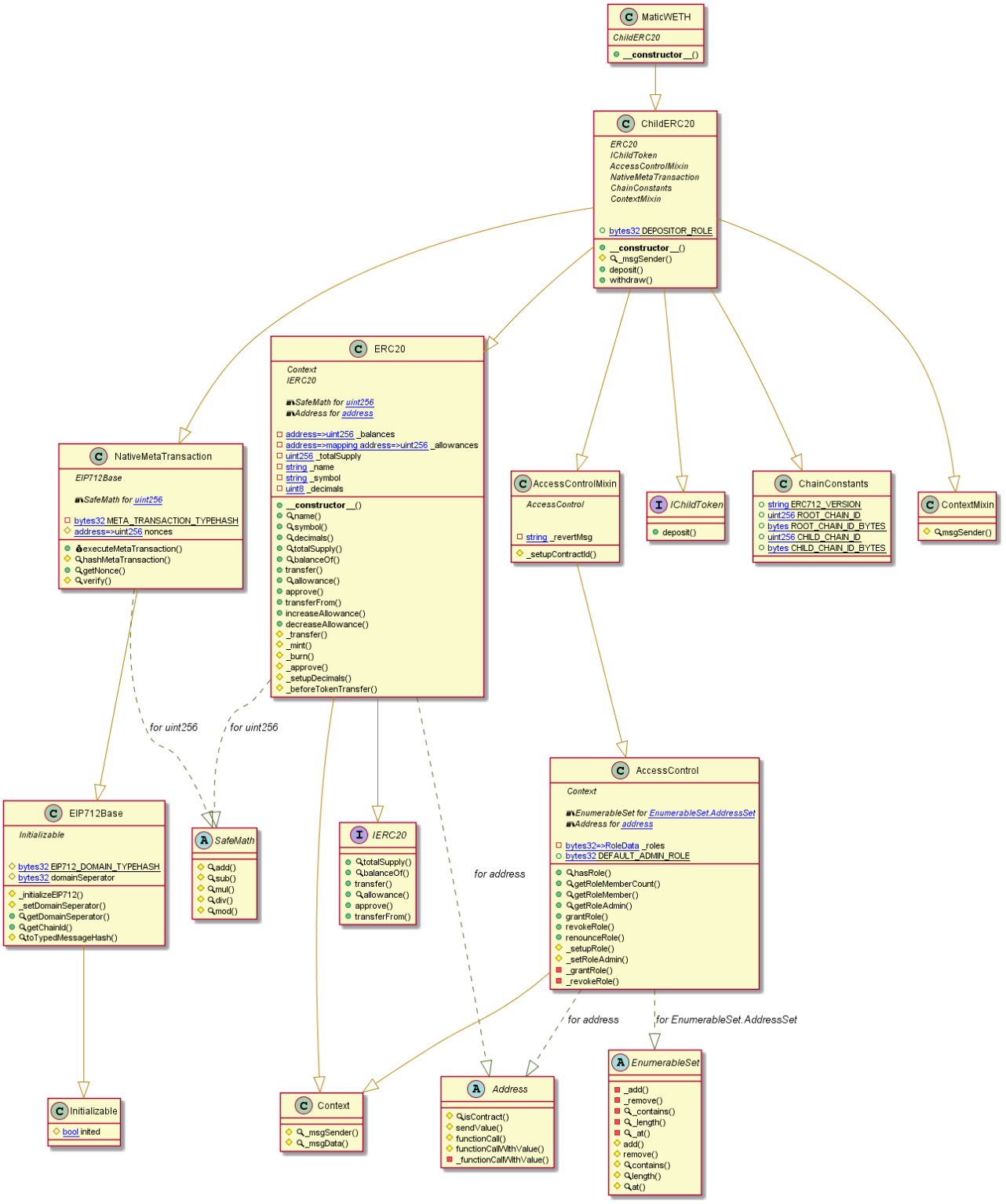
Due to the fact that the total number of test cases is unlimited, the audit makes no statements or warranties on the security of the code. It also cannot be considered as a sufficient assessment regarding the utility and safety of the code, bug-free status, or any other statements of the contract. While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

Appendix

Code Flow Diagram - Wrapped Ether 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 >> MaticWETH.sol

```
INFO:Detectors:
Contract locking ether found:
	Contract MaticWETH (MaticWETH.sol#1535-1538) has payable functions:
	 - NativeMetaTransaction.executeMetaTransaction(address,bytes,bytes32,bytes32,uint8)
(MaticWETH.sol#1344-1378)
	But does not have a function to withdraw the ether
Reference:
https://github.com/crytic/slither/wiki/Detector-Documentation#contracts-that-lock-ether
INFO:Detectors:
Version constraint ^0.6.0 contains known severe issues
(https://solidity.readthedocs.io/en/latest/bugs.html)
	- AbiReencodingHeadOverflowWithStaticArrayCleanup
	- DirtyBytesArrayToStorage
	- NestedCalldataArrayAbiReencodingSizeValidation
	- ABIDecodeTwoDimensionalArrayMemory
	- KeccakCaching
	- EmptyByteArrayCopy
	- DynamicArrayCleanup
	- MissingEscapingInFormatting
	- ArraySliceDynamicallyEncodedBaseType
	- ImplicitConstructorCallvalueCheck
	- TupleAssignmentMultiStackSlotComponents
	- MemoryArrayCreationOverflow
	- YulOptimizerRedundantAssignmentBreakContinue.
It is used by:
	- ^0.6.0 (MaticWETH.sol#12)
	- ^0.6.0 (MaticWETH.sol#39)
	- ^0.6.0 (MaticWETH.sol#119)
	- ^0.6.0 (MaticWETH.sol#425)
	- ^0.6.0 (MaticWETH.sol#734)
	- ^0.6.0 (MaticWETH.sol#980)
```

```
solc-0.6.6 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:
Redundant expression "this (MaticWETH.sol#30)" inContext (MaticWETH.sol#24-33)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#redundant-statements
INFO:Detectors:
executeMetaTransaction(address,bytes,bytes32,bytes32,uint8) should be declared external:
        - NativeMetaTransaction.executeMetaTransaction(address,bytes,bytes32,btes32,uint8)
(MaticWETH.sol#1344-1378)
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
INFO:Slither:MaticWETH.sol analyzed (16 contracts with 93 detectors), 20 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.

MaticWETH.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.
Pos: 397:4:

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.
Pos: 1446:12:

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: 1372:50:

Gas costs:
Gas requirement of function MaticWETH.withdraw 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: 1525:4:

Guard conditions:
Use "assert(x)" if you never ever want x to be false, not in any circumstance (apart from a bug in your code). Use "require(x)" if x can be false, due to e.g. invalid input or a failing external component.
Pos: 1147:8:
```

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.

MaticWETH.sol

```
Compiler version ^0.6.0 does not satisfy the ^0.5.8 semver
requirement
Pos: 1:11
Explicitly mark visibility of state
Pos: 5:1227
Avoid to use inline assembly. It is acceptable only in rare cases
Pos: 9:1287
Explicitly mark visibility of state
Pos: 5:1330
Error message for require is too long
Pos: 9:1356
Avoid to use low level calls.
Pos: 51:1371
Error message for require is too long
Pos: 9:1406
Avoid to use inline assembly. It is acceptable only in rare cases
Pos: 13:1445
Code contains empty blocks
Pos: 110:1535
```

Software analysis result:

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

Email: audit@EtherAuthority.io