Cyberscope

Audit Report

Merchant mWBTC

May 2024

Network	Merlin
Address	0x1a3fc43929AD6b931bBca8F66c520780a967fAAA
Audited by	© cyberscope

Table of Contents

Review

Contract Name	CErc20Immutable
Compiler Version	v0.8.20+commit.a1b79de6
Optimization	200 runs
Explorer	https://scan.merlinchain.io/address/0x1a3fc43929AD6b931bBca8F66c520780a967fAAA
Address	0x1a3fc43929AD6b931bBca8F66c520780a967fAAA
Network	Merlin
Name	Mer WBTC
Symbol	mWBTC
Decimals	8
Total Supply	2,200,000

Audit Updates

Initial Audit	09 May 2024

Source Files

Filename	SHA256
InterestRateModel.sol	8ca958179765a9ef12f955a76afdd6ac8bdacbe0be61216b6329e674b5739e7d

ExponentialNoError.sol	ef79b0e99297f924296b136e84d89861704 09eb233820b3e7733c2a6387707e9
ErrorReporter.sol	97062c28c271dac34dd5b46e74bfa31d6d 75f52b4c8c34653f275b2578a5e000
EIP20NonStandardInterface.sol	a2345b0d95fefe5a7256c2f7fb3bd3e1982 0b06d38c27235b8820116920a5894
EIP20Interface.sol	e1670bdc5381a0a06e605adf1d24226d25 132a43b560f2a190faf2d2cbf60aed
ComptrollerInterface.sol	98e442b63ace6598e6a4c46b539b5f2b41 a9fbe486c284e7e8acf11165b3be95
CTokenInterfaces.sol	7016d4ab7b6a30e571002f8172a38746b0 f3025d94935945f8cfa792a1d9920a
CToken.sol	c681be80c13f47d3ba2dd91b625e708246 8c19e0f846be8135cda9edca2e7390
CErc20Immutable.sol	8eb860c50e5eaffc5e916b1d64111228743 25bfb7e3d6b63909891ad342ca9ff
CErc20.sol	bb209daf7b40ddfc98d3606c019a5526d9 91cd7867ff65157783ca27e83be0b1

Overview

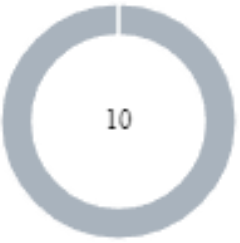
The Merchant CErc20Immutable contract is designed to create a money market for a specific EIP-20 compliant underlying asset. This contract extends the CErc20 contract, inheriting its functionalities related to handling a wrapped EIP-20 token within a decentralized lending platform. It focuses on providing an immutable version of the CToken, where key parameters such as the underlying asset address, comptroller address, interest rate model, and initial exchange rate are set at the time of deployment and cannot be altered afterward.

The main functionalities included in this contract involve initializing a new market with specific parameters defined at the time of deployment. These parameters include the underlying asset's address, the controlling mechanism (represented by the comptroller address), the model governing the interest rates, and the initial exchange rate between the native cToken and the underlying asset.

The contract constructor sets up the initial administrative privileges to the creator of the contract and transitions control to a designated administrator after the initial setup is complete. This transition ensures that the market's setup remains secure and under expected governance once operational.

By leveraging inheritance from the CErc20 contract, CErc20Immutable integrates the standard functionalities of an ERC-20 token while also embedding mechanisms typical of cTokens, such as the ability to accrue interest based on the supplied assets.

Findings Breakdown



Critical	0
Medium	0
Minor / Informative	10

Severity	Unresolved	Acknowledged	Resolved	Other
Critical	0	0	0	0
Medium	0	0	0	0
Minor / Informative	10	0	0	0

Diagnostics

Critical Medium Minor / Informative

Severity	Code	Description	Status
	COVC	Commented Out Verification Calls	Unresolved
	ISU	Inefficient Storage Use	Unresolved
	RVAC	Redundant Variables and Checks	Unresolved
	L04	Conformance to Solidity Naming Conventions	Unresolved
	L07	Missing Events Arithmetic	Unresolved
	L09	Dead Code Elimination	Unresolved
	L14	Uninitialized Variables in Local Scope	Unresolved
	L16	Validate Variable Setters	Unresolved
	L17	Usage of Solidity Assembly	Unresolved
	L19	Stable Compiler Version	Unresolved

COVC - Commented Out Verification Calls

Criticality	Minor / Informative
Location	CToken.sol#L131,575
Status	Unresolved

Description

The CToken contract exhibits instances of poor code practices, specifically involving commented out or missing verification function calls such as `transferVerify`, `mintVerify`, and `repayBorrowVerify`. This lack of clarity can lead to confusion. It is essential for code maintenance and updates that all modifications, especially those involving security checks or operational validations, are well-documented and clearly justified.

```
// comptroller.transferVerify(address(this), src, dst, tokens);
// comptroller.mintVerify(address(this), minter,
actualMintAmount, mintTokens);
```

Recommendation

To address this issue, there are a couple of approaches that can be taken. First, if these verification functions are likely to be necessary in future updates or are essential for potential compatibility with updates to the Comptroller, they should be reinstated with clear comments explaining their purpose and usage. Alternatively, if the commented out or removed verification calls are not intended to be used again, they should be completely removed from the code to prevent confusion.

ISU - Inefficient Storage Use

Criticality	Minor / Informative
Location	CToken.sol#L338 CErc20.sol#L161
Status	Unresolved

Description

Inefficient use of storage slots are present in several functions, leading to unnecessary gas consumption during execution. Notably, the `doTransferIn` function repeatedly reads the same underlying state variable three times instead of loading it once onto the execution stack for repeated use. Additionally, a storage variable `borrowSnapshot` is declared where a memory type could be utilized.

```
function borrowBalanceStoredInternal(address account)
    internal
    view
    returns (uint256)
{
    /* Get borrowBalance and borrowIndex */
    BorrowSnapshot storage borrowSnapshot =
accountBorrows[account];

    /* If borrowBalance = 0 then borrowIndex is likely also 0.
    * Rather than failing the calculation with a division by 0,
we immediately return 0 in this case.
    */
    if (borrowSnapshot.principal == 0) {
        return 0;
    }

    /* Calculate new borrow balance using the interest index:
    *  recentBorrowBalance = borrower.borrowBalance *
market.borrowIndex / borrower.borrowIndex
    */
    uint256 principalTimesIndex = borrowSnapshot.principal *
borrowIndex;
    return principalTimesIndex / borrowSnapshot.interestIndex;
}
```

Recommendation

Optimizations should be implemented to reduce the frequency of storage reads by loading variables into memory or onto the stack a single time and reusing them throughout the function's logic.

RVAC - Redundant Variables and Checks

Criticality	Minor / Informative
Location	CToken.sol#L80
Status	Unresolved

Description

There are instances of unnecessary local variables and validations that do not contribute value to the function's operations. These intermediate steps consume additional gas and complicate the code structure, which detracts from the overall readability and efficiency of the function.

```
function transferTokens(
        address spender,
        address src,
        address dst,
        uint256 tokens
    ) internal returns (uint256) {
        /* Fail if transfer not allowed */
        uint256 allowed = comptroller.transferAllowed(
            address(this),
            src,
            dst,
            tokens
        );
        if (allowed != 0) {
            revert TransferComptrollerRejection(allowed);
        }

        /* Do not allow self-transfers */
        if (src == dst) {
            revert TransferNotAllowed();
        }

        /* Get the allowance, infinite for the account owner */
        uint256 startingAllowance = 0;
        if (spender == src) {
            startingAllowance = type(uint256).max;
        } else {
            startingAllowance =
transferAllowances[src][spender];
        }

        /* Do the calculations, checking for {under,over}flow
*/
        uint256 allowanceNew = startingAllowance - tokens;
        uint256 srcTokensNew = accountTokens[src] - tokens;
        uint256 dstTokensNew = accountTokens[dst] + tokens;

        ////////////////////////
        // EFFECTS & INTERACTIONS
        // (No safe failures beyond this point)

        accountTokens[src] = srcTokensNew;
        accountTokens[dst] = dstTokensNew;

        /* Eat some of the allowance (if necessary) */
        if (startingAllowance != type(uint256).max) {
            transferAllowances[src][spender] = allowanceNew;
        }

        /* We emit a Transfer event */
```

```
        emit Transfer(src, dst, tokens);

        // unused function
        // comptroller.transferVerify(address(this), src, dst,
tokens);

        return NO_ERROR;
    }
```

Recommendation

It is recommended to eliminate redundant local variable declarations and integrating validations directly with the operational logic. Removing intermediate steps will not only reduce the gas cost associated with these operations but also enhance the clarity and maintainability of the code.

L04 - Conformance to Solidity Naming Conventions

Criticality	Minor / Informative
Location	CToken.sol#L1092,1119,1146,1174,1287,1354 CErc20.sol#L136,224
Status	Unresolved

Description

The Solidity style guide is a set of guidelines for writing clean and consistent Solidity code. Adhering to a style guide can help improve the readability and maintainability of the Solidity code, making it easier for others to understand and work with.

The followings are a few key points from the Solidity style guide:

1. Use camelCase for function and variable names, with the first letter in lowercase (e.g., myVariable, updateCounter).
2. Use PascalCase for contract, struct, and enum names, with the first letter in uppercase (e.g., MyContract, UserStruct, ErrorEnum).
3. Use uppercase for constant variables and enums (e.g., MAX_VALUE, ERROR_CODE).
4. Use indentation to improve readability and structure.
5. Use spaces between operators and after commas.
6. Use comments to explain the purpose and behavior of the code.
7. Keep lines short (around 120 characters) to improve readability.

```
function _setPendingAdmin(address payable newPendingAdmin)
        external
        override
        returns (uint256)
    {
        // Check caller = admin
...
        pendingAdmin = newPendingAdmin;

        // Emit NewPendingAdmin(oldPendingAdmin,
newPendingAdmin)
        emit NewPendingAdmin(oldPendingAdmin, newPendingAdmin);

        return NO_ERROR;
    }

...
```

Recommendation

By following the Solidity naming convention guidelines, the codebase increased the readability, maintainability, and makes it easier to work with.
Find more information on the Solidity documentation
https://docs.soliditylang.org/en/v0.8.17/style-guide.html#naming-convention.

L07 - Missing Events Arithmetic

Criticality	Minor / Informative
Location	CToken.sol#L45
Status	Unresolved

Description

Events are a way to record and log information about changes or actions that occur within a contract. They are often used to notify external parties or clients about events that have occurred within the contract, such as the transfer of tokens or the completion of a task.

It's important to carefully design and implement the events in a contract, and to ensure that all required events are included. It's also a good idea to test the contract to ensure that all events are being properly triggered and logged.

```
initialExchangeRateMantissa = initialExchangeRateMantissa_
```

Recommendation

By including all required events in the contract and thoroughly testing the contract's functionality, the contract ensures that it performs as intended and does not have any missing events that could cause issues with its arithmetic.

L09 - Dead Code Elimination

Criticality	Minor / Informative
Location	CToken.sol#L509,521,583,594,610,700,710,770,781,797,871,901,1225,1243
Status	Unresolved

Description

In Solidity, dead code is code that is written in the contract, but is never executed or reached during normal contract execution. Dead code can occur for a variety of reasons, such as:

- Conditional statements that are always false.
- Functions that are never called.
- Unreachable code (e.g., code that follows a return statement).

Dead code can make a contract more difficult to understand and maintain, and can also increase the size of the contract and the cost of deploying and interacting with it.

```
function mintInternal(uint256 mintAmount) internal nonReentrant
{
        accrueInterest();
        // mintFresh emits the actual Mint event if successful
and logs on errors, so we don't need to
        mintFresh(msg.sender, mintAmount);
    }

...
```

Recommendation

To avoid creating dead code, it's important to carefully consider the logic and flow of the contract and to remove any code that is not needed or that is never executed. This can help improve the clarity and efficiency of the contract.

L14 - Uninitialized Variables in Local Scope

Criticality	Minor / Informative
Location	CToken.sol#L1249
Status	Unresolved

Description

Using an uninitialized local variable can lead to unpredictable behavior and potentially cause errors in the contract. It's important to always initialize local variables with appropriate values before using them.

```
256 actualAddAmount;
```

Recommendation

By initializing local variables before using them, the contract ensures that the functions behave as expected and avoid potential issues.

L16 - Validate Variable Setters

Criticality	Minor / Informative
Location	CToken.sol#L1106 CErc20Immutable.sol#L40 CErc20.sol#L37
Status	Unresolved

Description

The contract performs operations on variables that have been configured on user-supplied input. These variables are missing of proper check for the case where a value is zero. This can lead to problems when the contract is executed, as certain actions may not be properly handled when the value is zero.

```
pendingAdmin = newPendingAdmin
admin = admin_
underlying = underlying_
```

Recommendation

By adding the proper check, the contract will not allow the variables to be configured with zero value. This will ensure that the contract can handle all possible input values and avoid unexpected behavior or errors. Hence, it can help to prevent the contract from being exploited or operating unexpectedly.

L17 - Usage of Solidity Assembly

Criticality	Minor / Informative
Location	CErc20.sol#L169,203
Status	Unresolved

Description

Using assembly can be useful for optimizing code, but it can also be error-prone. It's important to carefully test and debug assembly code to ensure that it is correct and does not contain any errors.

Some common types of errors that can occur when using assembly in Solidity include Syntax, Type, Out-of-bounds, Stack, and Revert.

```
assembly {
            switch returndatasize()
                case 0 {                        // This is a
non-standard ERC-20
                    success := not(0)           // set success
to true
                }
                case 32 {                       // This is a
compliant ERC-20
                    returndatacopy(0, 0, 32)
                    success := mload(0)         // Set `success
= returndata` of override external call
                }
                default {                       // This is an
excessively non-compliant ERC-20, revert.
                    revert(0, 0)
                }
        }

...
```

Recommendation

It is recommended to use assembly sparingly and only when necessary, as it can be difficult to read and understand compared to Solidity code.

L19 - Stable Compiler Version

Criticality	Minor / Informative
Location	CToken.sol#L2 CErc20Immutable.sol#L2 CErc20.sol#L2
Status	Unresolved

Description

The `^` symbol indicates that any version of Solidity that is compatible with the specified version (i.e., any version that is a higher minor or patch version) can be used to compile the contract. The version lock is a mechanism that allows the author to specify a minimum version of the Solidity compiler that must be used to compile the contract code. This is useful because it ensures that the contract will be compiled using a version of the compiler that is known to be compatible with the code.

```
pragma solidity ^0.8.10;
```

Recommendation

The team is advised to lock the pragma to ensure the stability of the codebase. The locked pragma version ensures that the contract will not be deployed with an unexpected version. An unexpected version may produce vulnerabilities and undiscovered bugs. The compiler should be configured to the lowest version that provides all the required functionality for the codebase. As a result, the project will be compiled in a well-tested LTS (Long Term Support) environment.

Functions Analysis

Contract	Type	Bases		
	Function Name	**Visibility**	**Mutability**	**Modifiers**
CToken	Implementation	CTokenInterface, Exponential NoError, TokenErrorReporter		
	initialize	Public	✓	-
	transferTokens	Internal	✓	
	transfer	External	✓	nonReentrant
	transferFrom	External	✓	nonReentrant
	approve	External	✓	-
	allowance	External		-
	balanceOf	External		-
	balanceOfUnderlying	External	✓	-
	getAccountSnapshot	External		-
	getBlockNumber	Internal		
	borrowRatePerBlock	External		-
	supplyRatePerBlock	External		-
	totalBorrowsCurrent	External	✓	nonReentrant
	borrowBalanceCurrent	External	✓	nonReentrant
	borrowBalanceStored	Public		-
	borrowBalanceStoredInternal	Internal		

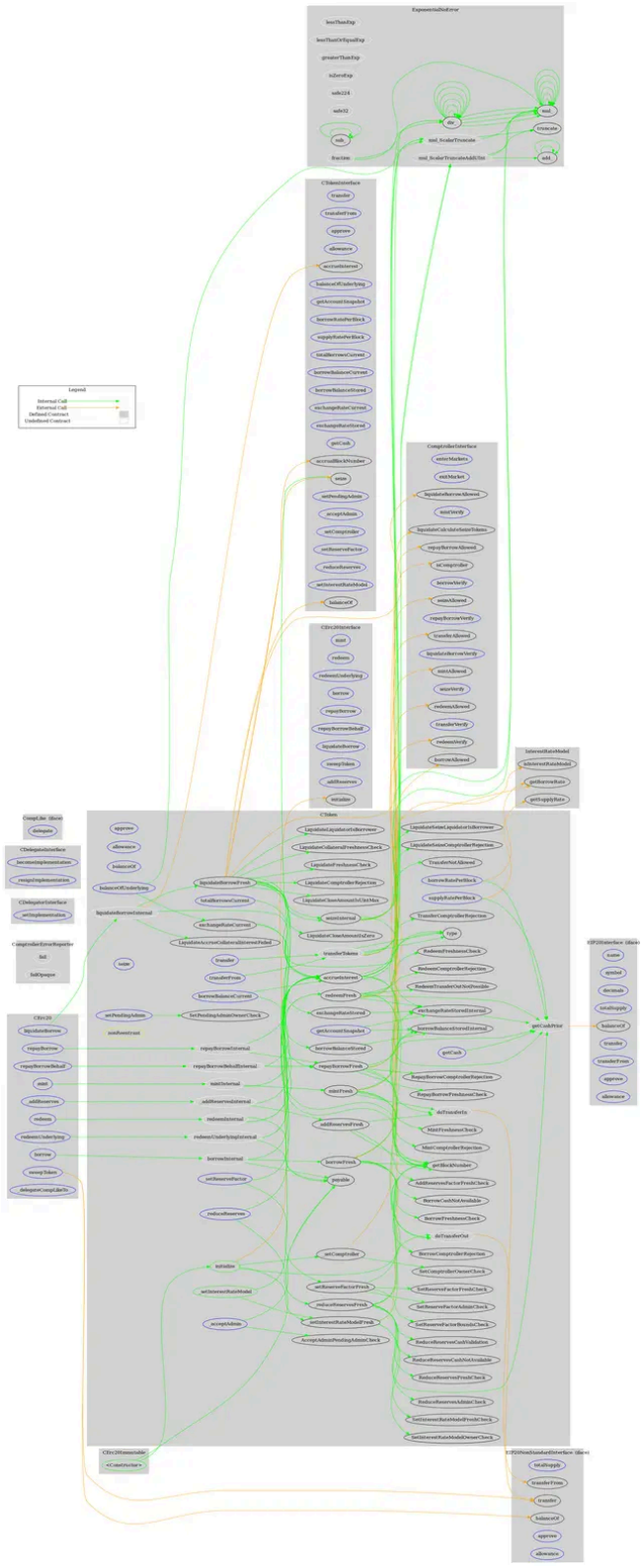
	exchangeRateCurrent	Public	✓	nonReentrant
	exchangeRateStored	Public		-
	exchangeRateStoredInternal	Internal		
	getCash	External		-
	accrueInterest	Public	✓	-
	mintInternal	Internal	✓	nonReentrant
	mintFresh	Internal	✓	
	redeemInternal	Internal	✓	nonReentrant
	redeemUnderlyingInternal	Internal	✓	nonReentrant
	redeemFresh	Internal	✓	
	borrowInternal	Internal	✓	nonReentrant
	borrowFresh	Internal	✓	
	repayBorrowInternal	Internal	✓	nonReentrant
	repayBorrowBehalfInternal	Internal	✓	nonReentrant
	repayBorrowFresh	Internal	✓	
	liquidateBorrowInternal	Internal	✓	nonReentrant
	liquidateBorrowFresh	Internal	✓	
	seize	External	✓	nonReentrant
	seizeInternal	Internal	✓	
	_setPendingAdmin	External	✓	-
	_acceptAdmin	External	✓	-
	_setComptroller	Public	✓	-
	_setReserveFactor	External	✓	nonReentrant

	_setReserveFactorFresh	Internal	✓	
	_addReservesInternal	Internal	✓	nonReentrant
	_addReservesFresh	Internal	✓	
	_reduceReserves	External	✓	nonReentrant
	_reduceReservesFresh	Internal	✓	
	_setInterestRateModel	Public	✓	-
	_setInterestRateModelFresh	Internal	✓	
	getCashPrior	Internal		
	doTransferIn	Internal	✓	
	doTransferOut	Internal	✓	
CErc20Immutable	Implementation	CErc20		
		Public	✓	-
CompLike	Interface			
	delegate	External	✓	-
CErc20	Implementation	CToken, CErc20Interface		
	initialize	Public	✓	-
	mint	External	✓	-
	redeem	External	✓	-
	redeemUnderlying	External	✓	-
	borrow	External	✓	-

	repayBorrow	External	✓	-
	repayBorrowBehalf	External	✓	-
	liquidateBorrow	External	✓	-
	sweepToken	External	✓	-
	_addReserves	External	✓	-
	getCashPrior	Internal		
	doTransferIn	Internal	✓	
	doTransferOut	Internal	✓	
	_delegateCompLikeTo	External	✓	-

Inheritance Graph

Flow Graph



Summary

Merchant mWBTC contract implements a token mechanism. This audit investigates security issues, business logic concerns and potential improvements.

Disclaimer

The information provided in this report does not constitute investment, financial or trading advice and you should not treat any of the document's content as such. This report may not be transmitted, disclosed, referred to or relied upon by any person for any purposes nor may copies be delivered to any other person other than the Company without Cyberscope's prior written consent. This report is not nor should be considered an "endorsement" or "disapproval" of any particular project or team. This report is not nor should be regarded as an indication of the economics or value of any "product" or "asset" created by any team or project that contracts Cyberscope to perform a security assessment. This document does not provide any warranty or guarantee regarding the absolute bug-free nature of the technology analyzed, nor do they provide any indication of the technologies proprietors' business, business model or legal compliance. This report should not be used in any way to make decisions around investment or involvement with any particular project. This report represents an extensive assessment process intending to help our customers increase the quality of their code while reducing the high level of risk presented by cryptographic tokens and blockchain technology.

Blockchain technology and cryptographic assets present a high level of ongoing risk Cyberscope's position is that each company and individual are responsible for their own due diligence and continuous security Cyberscope's goal is to help reduce the attack vectors and the high level of variance associated with utilizing new and consistently changing technologies and in no way claims any guarantee of security or functionality of the technology we agree to analyze. The assessment services provided by Cyberscope are subject to dependencies and are under continuing development. You agree that your access and/or use including but not limited to any services reports and materials will be at your sole risk on an as-is where-is and as-available basis Cryptographic tokens are emergent technologies and carry with them high levels of technical risk and uncertainty. The assessment reports could include false positives false negatives and other unpredictable results. The services may access and depend upon multiple layers of third parties.

About Cyberscope

Cyberscope is a blockchain cybersecurity company that was founded with the vision to make web3.0 a safer place for investors and developers. Since its launch, it has worked with thousands of projects and is estimated to have secured tens of millions of investors' funds.

Cyberscope is one of the leading smart contract audit firms in the crypto space and has built a high-profile network of clients and partners.

The Cyberscope team

https://www.cyberscope.io