# Coin98DollarMintBurn

## Smart Contract Audit Report
## Prepared for Coin98


| | |
|---|---|
| **Date Issued:** | May 27, 2022 |
| **Project ID:** | AUDIT2022032-2 |
| **Version:** | v1.0 |
| **Confidentiality Level:** | Public |

## Report Information

| | |
|---|---|
| **Project ID** | AUDIT2022032-2 |
| **Version** | v1.0 |
| **Client** | Coin98 |
| **Project** | Coin98DollarMintBurn |
| **Auditor(s)** | Peeraphut Punsuwan<br>Ronnachai Chaipha<br>Sorawish Laovakul |
| **Author(s)** | Wachirawit Kanpanluk |
| **Reviewer** | Patipon Suwanbol |
| **Confidentiality Level** | Public |

## Version History

| Version | Date | Description | Author(s) |
|---|---|---|---|
| 1.0 | May 27, 2022 | Full report | Wachirawit Kanpanluk |

## Contact Information



| | |
|---|---|
| **Company** | Inspex |
| **Phone** | (+66) 90 888 7186 |
| **Telegram** | t.me/inspexco |
| **Email** | audit@inspex.co |

# Table of Contents

# 1. Executive Summary

As requested by Coin98 , Inspex team conducted an audit to verify the security posture of the Coin98DollarMintBurn smart contracts between May 18, 2022 and May 19, 2022. During the audit, Inspex team examined all smart contracts and the overall operation within the scope to understand the overview of Coin98DollarMintBurn smart contracts. Static code analysis, dynamic analysis, and manual review were done in conjunction to identify smart contract vulnerabilities together with technical & business logic flaws that may be exposed to the potential risk of the platform and the ecosystem. Practical recommendations are provided according to each vulnerability found and should be followed to remediate the issue.

## 1.1. Audit Result

In the initial audit, Inspex found 2 high, 1 medium, 1 low, 1 very low, and 1 info-severity issues. With the project team's prompt response, 1 medium, 1 low, 1 very low and 1 info-severity issues were resolved in the reassessment, while 2 high-severity issues were acknowledged by the team. However, in the long run, Inspex suggests resolving all issues found in this report.

## 1.2. Disclaimer

This security audit is not produced to supplant any other type of assessment and does not guarantee the discovery of all security vulnerabilities within the scope of the assessment. However, we warrant that this audit is conducted with goodwill, professional approach, and competence. Since an assessment from one single party cannot be confirmed to cover all possible issues within the smart contract(s), Inspex suggests conducting multiple independent assessments to minimize the risks. Lastly, nothing contained in this audit report should be considered as investment advice.

# 2. Project Overview

## 2.1. Project Introduction

Coin98 is a Financial Services Builder that creates and develops an ecosystem of DeFi protocols, applications, and NFTs on multiple blockchains. The platform can help people to access DeFi services effortlessly.

Coin98DollarMintBurn is the contract that provides the mint / burn $cUSD mechanism to users. The users can mint the $cUSD by transferring the tokens to the contract as a collateral. The asset tokens will be transferred back when users burn the $cUSD.

**Scope Information:**

| | |
|---|---|
| **Project Name** | Coin98DollarMintBurn |
| **Website** | https://coin98.com/ |
| **Smart Contract Type** | Ethereum Smart Contract |
| **Chain** | BNB Smart Chain |
| **Programming Language** | Solidity |
| **Category** | Token, Stable Coin |

**Audit Information:**

| | |
|---|---|
| **Audit Method** | Whitebox |
| **Audit Date** | May 18, 2022 - May 19, 2022 |
| **Reassessment Date** | May 26, 2022 |

The audit method can be categorized into two types depending on the assessment targets provided:

1. **Whitebox**: The complete source code of the smart contracts are provided for the assessment.
2. **Blackbox**: Only the bytecodes of the smart contracts are provided for the assessment.

## 2.2. Scope

The following smart contracts were audited and reassessed by Inspex in detail:

**Initial Audit: (Commit: 303cc426c82dea83892b51a68d1bab09022bf754)**

| Contract | Location (URL) |
| --- | --- |
| Coin98DollarMintBurn | https://github.com/coin98/coin98-dollar-mint-burn/blob/303cc426c8/contracts/Coin98DollarMintBurn.sol |

**Reassessment: (Commit: 40d70294a12bbc9692fc47195a3a5bc488b9f89f)**

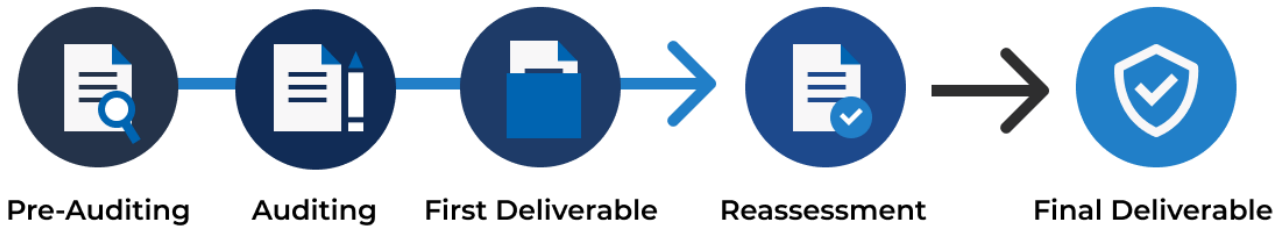
| Contract | Location (URL) |
| --- | --- |
| Coin98DollarMintBurn | https://github.com/coin98/coin98-dollar-mint-burn/blob/40d70294a1/contracts/Coin98DollarMintBurn.sol |

The assessment scope covers only the in-scope smart contracts and the smart contracts that they inherit from.

# 3. Methodology

Inspex conducts the following procedure to enhance the security level of our clients' smart contracts:

1. **Pre-Auditing**: Getting to understand the overall operations of the related smart contracts, checking for readiness, and preparing for the auditing
2. **Auditing**: Inspecting the smart contracts using automated analysis tools and manual analysis by a team of professionals
3. **First Deliverable and Consulting**: Delivering a preliminary report on the findings with suggestions on how to remediate those issues and providing consultation
4. **Reassessment**: Verifying the status of the issues and whether there are any other complications in the fixes applied
5. **Final Deliverable**: Providing a full report with the detailed status of each issue




## 3.1. Test Categories

Inspex smart contract auditing methodology consists of both automated testing with scanning tools and manual testing by experienced testers. We have categorized the tests into 3 categories as follows:

1. **General Smart Contract Vulnerability (General)** - Smart contracts are analyzed automatically using static code analysis tools for general smart contract coding bugs, which are then verified manually to remove all false positives generated.
2. **Advanced Smart Contract Vulnerability (Advanced)** - The workflow, logic, and the actual behavior of the smart contracts are manually analyzed in-depth to determine any flaws that can cause technical or business damage to the smart contracts or the users of the smart contracts.
3. **Smart Contract Best Practice (Best Practice)** - The code of smart contracts is then analyzed from the development perspective, providing suggestions to improve the overall code quality using standardized best practices.

## 3.2. Audit Items

The testing items checked are based on our Smart Contract Security Testing Guide (SCSTG) v1.0 (https://github.com/InspexCo/SCSTG/releases/download/v1.0/SCSTG_v1.0.pdf) which covers most prevalent risks in smart contracts. The latest version of the document can also be found at https://inspex.gitbook.io/testing-guide/.

The following audit items were checked during the auditing activity:

| Testing Category | Testing Items |
|---|---|
| 1. Architecture and Design | 1.1. Proper measures should be used to control the modifications of smart contract logic<br>1.2. The latest stable compiler version should be used<br>1.3. The circuit breaker mechanism should not prevent users from withdrawing their funds<br>1.4. The smart contract source code should be publicly available<br>1.5. State variables should not be unfairly controlled by privileged accounts<br>1.6. Least privilege principle should be used for the rights of each role |
| 2. Access Control | 2.1. Contract self-destruct should not be done by unauthorized actors<br>2.2. Contract ownership should not be modifiable by unauthorized actors<br>2.3. Access control should be defined and enforced for each actor roles<br>2.4. Authentication measures must be able to correctly identify the user<br>2.5. Smart contract initialization should be done only once by an authorized party<br>2.6. tx.origin should not be used for authorization |
| 3. Error Handling and Logging | 3.1. Function return values should be checked to handle different results<br>3.2. Privileged functions or modifications of critical states should be logged<br>3.3. Modifier should not skip function execution without reverting |
| 4. Business Logic | 4.1. The business logic implementation should correspond to the business design<br>4.2. Measures should be implemented to prevent undesired effects from the ordering of transactions<br>4.3. msg.value should not be used in loop iteration |
| 5. Blockchain Data | 5.1. Result from random value generation should not be predictable<br>5.2. Spot price should not be used as a data source for price oracles<br>5.3. Timestamp should not be used to execute critical functions<br>5.4. Plain sensitive data should not be stored on-chain<br>5.5. Modification of array state should not be done by value<br>5.6. State variable should not be used without being initialized |

| Testing Category | Testing Items |
| --- | --- |
| 6. External Components | 6.1. Unknown external components should not be invoked<br>6.2. Funds should not be approved or transferred to unknown accounts<br>6.3. Reentrant calling should not negatively affect the contract states<br>6.4. Vulnerable or outdated components should not be used in the smart contract<br>6.5. Deprecated components that have no longer been supported should not be used in the smart contract<br>6.6. Delegatecall should not be used on untrusted contracts |
| 7. Arithmetic | 7.1. Values should be checked before performing arithmetic operations to prevent overflows and underflows<br>7.2. Explicit conversion of types should be checked to prevent unexpected results<br>7.3. Integer division should not be done before multiplication to prevent loss of precision |
| 8. Denial of Services | 8.1. State changing functions that loop over unbounded data structures should not be used<br>8.2. Unexpected revert should not make the whole smart contract unusable<br>8.3. Strict equalities should not cause the function to be unusable |
| 9. Best Practices | 9.1. State and function visibility should be explicitly labeled<br>9.2. Token implementation should comply with the standard specification<br>9.3. Floating pragma version should not be used<br>9.4. Builtin symbols should not be shadowed<br>9.5. Functions that are never called internally should not have public visibility<br>9.6. Assert statement should not be used for validating common conditions |

## 3.3. Risk Rating

OWASP Risk Rating Methodology (https://owasp.org/www-community/OWASP_Risk_Rating_Methodology) is used to determine the severity of each issue with the following criteria:

- **Likelihood**: a measure of how likely this vulnerability is to be uncovered and exploited by an attacker
- **Impact**: a measure of the damage caused by a successful attack

Both likelihood and impact can be categorized into three levels: **Low**, **Medium**, and **High**.

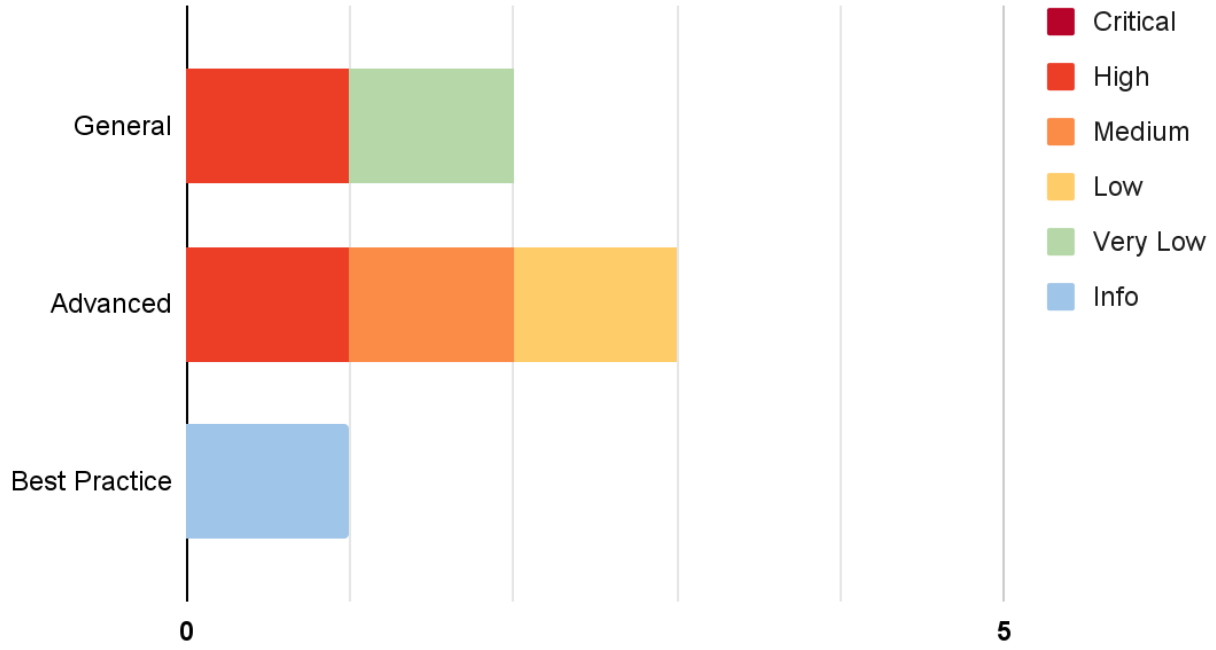
**Severity** is the overall risk of the issue. It can be categorized into five levels: **Very Low**, **Low**, **Medium**, **High**, and **Critical**. It is calculated from the combination of likelihood and impact factors using the matrix below. The severity of findings with no likelihood or impact would be categorized as **Info**.

| Impact \ Likelihood | Low | Medium | High |
|---|---|---|---|
| Low | Very Low | Low | Medium |
| Medium | Low | Medium | High |
| High | Medium | High | Critical |

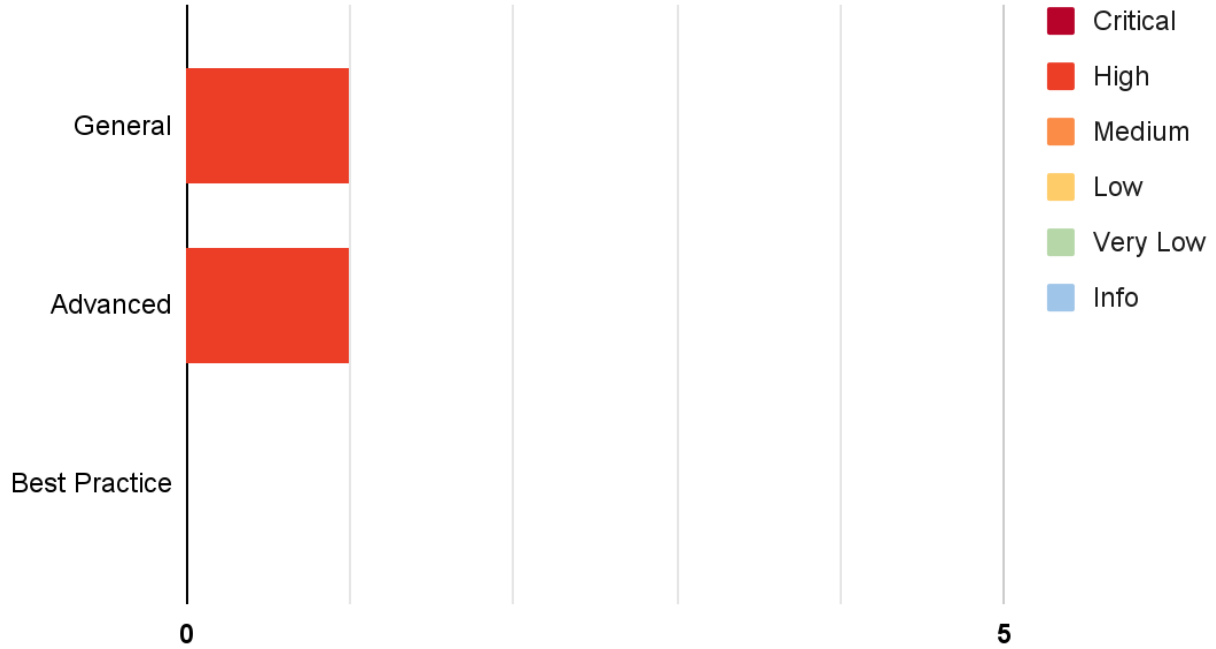
# 4. Summary of Findings

The following charts show the number of the issues found during the assessment and the issues acknowledged in the reassessment, categorized into three categories: **General**, **Advanced**, and **Best Practice**.

**Assessment:**




**Reassessment:**



The statuses of the issues are defined as follows:

| Status | Description |
|---|---|
| Resolved | The issue has been resolved and has no further complications. |
| Resolved * | The issue has been resolved with mitigations and clarifications. For the clarification or mitigation detail, please refer to Chapter 5. |
| Acknowledged | The issue's risk has been acknowledged and accepted. |
| No Security Impact | The best practice recommendation has been acknowledged. |

The information and status of each issue can be found in the following table:

| ID | Title | Category | Severity | Status |
|---|---|---|---|---|
| IDX-001 | Design Flaw in cUSD Token | Advanced | High | Acknowledged |
| IDX-002 | Centralized Control of State Variable | General | High | Acknowledged |
| IDX-003 | Incorrect Daily Total Mint and Burn Amount Update | Advanced | Medium | Resolved |
| IDX-004 | Division Before Multiplication | Advanced | Low | Resolved |
| IDX-005 | Insufficient Logging for Privileged Functions | General | Very Low | Resolved |
| IDX-006 | Improper Function Visibility | Best Practice | Info | Resolved |

* The mitigations or clarifications by Coin98 can be found in Chapter 5.

# 5. Detailed Findings Information

## 5.1. Design Flaw in cUSD Token

| | |
|---|---|
| **ID** | IDX-001 |
| **Target** | Coin98DollarMintBurn |
| **Category** | Advanced Smart Contract Vulnerability |
| **CWE** | CWE-701: Weaknesses Introduced During Design |
| **Risk** | **Severity:** High<br><br>**Impact:** High<br>When the reserve in the `Coin98DollarMintBurn` contract is insufficient, the users cannot burn the $cUSD to swap the asset token back.<br><br>**Likelihood:** Medium<br>When the asset's price falls, the reserve is likely to be insufficient for all users. However, the `burn()` function has a daily burn limit, which can delay the dumping of the swap $cUSD token. |
| **Status** | **Acknowledged**<br>The Coin98 team has acknowledged and accepted the issue's risk and has confirmed that it has prepared a pool for swapping in case the contract's funds are insufficient. |

### 5.1.1. Description

The user can use the `mint()` function that will transfer the asset tokens from the user to the contract, and then the $cUSD will be minted to the user for the same amount as the asset value in USD.

**Coin98DollarMintBurn.sol**

```
1168 function mint(uint256 _id, uint256 amount) public onlyActiveMinter(_id) {
1169     require(
1170         amount > 0,
1171         "Coin98DollarMintBurn: Amount must be a positive number and greater
     than zero"
1172     );
1173     TokenMinter storage minter = TokenMinters[_id];
1174
1175     uint256 currentTotalMintedToday = checkTotalMinted(minter, amount);
1176
1177     uint256[] memory amountToTransfer = new uint256[](minter.pairs.length);
1178
1179     for (uint256 i = 0; i < minter.pairs.length; i++) {
1180         uint256 tokenDecimals = minter.decimals[i];
1181         uint256 valueByPercent = amount.mul(minter.percents[i]).div(
```

```
1182            Percent
1183        );
1184
1185        // Feed the latest price by ChainLink
1186        (uint256 price, uint256 priceDecimals) = getLatestPrice(
1187            minter.priceFeed[i]
1188        );
1189
1190        uint256 amountToBurn = valueByPercent
1191            .mul(10**tokenDecimals)
1192            .div(price)
1193            .mul(10**priceDecimals)
1194            .div(BASE_DECIMALS);
1195
1196        // Transfer money first before do anything effects
1197        IERC20(minter.pairs[i]).safeTransferFrom(
1198            msg.sender,
1199            address(this),
1200            amountToBurn
1201        );
1202        amountToTransfer[i] = amountToBurn;
1203    }
1204
1205    uint256 systemFee = amount.mul(minter.systemFee).div(Percent);
1206    // Update tracking information
1207    minter.totalSystemFee = minter.totalSystemFee.add(systemFee);
1208
1209    // Claim system fee for each exchange cUSD
1210    amount = amount.sub(systemFee);
1211
1212    require(
1213        amount > 0,
1214        "Coin98DollarMintBurn: Total Mint must be a positive number and greater
    than zero"
1215    );
1216
1217    // Mint CUSD Token to .sender
1218    CUSD_TOKEN.mint(msg.sender, amount);
1219
1220    emit Mint(
1221        _id,
1222        msg.sender,
1223        amountToTransfer,
1224        amount,
1225        systemFee,
1226        currentTotalMintedToday
1227    );
```

```
1228 }
```

The user can execute the `burn()` function to burn the $cUSD, then the asset tokens with the same value in USD will be transferred to the user.

**Coin98DollarMintBurn.sol**

```
1112 function burn(uint256 _id, uint256 amount) public onlyActiveBurner(_id) {
1113     require(
1114         amount > 0,
1115         "Coin98DollarMintBurn: Amount must be a positive number and greater
     than zero"
1116     );
1117     TokenBurner storage burner = TokenBurners[_id];
1118
1119     (uint256 price, uint256 priceDecimals) = getLatestPrice(
1120         burner.priceFeed
1121     );
1122
1123     uint256 amountToBurn = amount
1124         .mul(price)
1125         .mul(BASE_DECIMALS)
1126         .div(10**priceDecimals)
1127         .div(10**burner.decimals);
1128
1129     uint256 currentTotalBurnedToday = checkTotalBurned(
1130         burner,
1131         amountToBurn
1132     );
1133
1134     // Transfer money first before do anything effects
1135     CUSD_TOKEN.safeTransferFrom(msg.sender, address(this), amountToBurn);
1136     CUSD_TOKEN.burn(amountToBurn);
1137
1138     // Update tracking information
1139     uint256 systemFee = amount.mul(burner.systemFee).div(Percent);
1140     burner.totalSystemFee = burner.totalSystemFee.add(systemFee);
1141
1142     IERC20 tokenMint = IERC20(burner.token);
1143     uint256 amountToMint = amount.sub(systemFee);
1144
1145     require(
1146         amountToMint > 0 &&
1147             tokenMint.balanceOf(address(this)) >= amountToMint,
1148         "Coin98DollarMintBurn: Not enough balance to mint or invalid amount"
1149     );
1150
1151     tokenMint.safeTransfer(msg.sender, amountToMint);
```

```
1152
1153        emit Burn(
1154            _id,
1155            msg.sender,
1156            amountToBurn,
1157            amountToMint,
1158            systemFee,
1159            currentTotalBurnedToday
1160        );
1161    }
```

In case the reserve collateral is not enough, the user cannot burn the $cUSD to get the asset back at line 1145, for example:

The user transfers the $C98 in 10$ and the $BUSD in 90$ to mint the 100 $cUSD; however, the user cannot burn the 100 $cUSD to get the asset tokens back because the $BUSD is insufficient in the contract. In case the owner does not transfer the sufficient reserve to the contract. This results in the users being unable to burn the $cUSD to get the asset tokens back.

### 5.1.2. Remediation

Inspex suggests implementing a mechanism to prevent the platform from running out of collateral reserves.

- Fully asset-backed with stable coin
  - The $cUSD should be backed by the stable coin with a 1:1 ratio to confirm that the reserve assets will always be sufficient for the users
- Over collateral with liquidation mechanism
  - The $cUSD should be backed by the collateral asset which is worth more than the $cUSD minted amount in USD. The Over collateral mechanism must be implemented along with the liquidation mechanism for preventing the bad debt for the platform.

## 5.2. Centralized Control of State Variable

| | |
|---|---|
| **ID** | IDX-002 |
| **Target** | Coin98DollarMintBurn |
| **Category** | General Smart Contract Vulnerability |
| **CWE** | CWE-284: Improper Access Control |
| **Risk** | **Severity: High**<br><br>**Impact: High**<br>The controlling authorities can change the critical state variables to gain additional profit. Thus, it is unfair to the other users. For example, the owner can withdraw all user's deposited funds from the `Coin98DollarMintBurn` contract.<br><br>**Likelihood: Medium**<br>There is nothing to restrict the changes from being done; however, this action can only be done by the privileged roles. |
| **Status** | **Acknowledged**<br>The Coin98 team has acknowledged and accepted the issue's risk and has confirmed that the user's fund will be minimally impacted. |

### 5.2.1. Description

Critical state variables can be updated at any time by the controlling authorities. Changes in these variables can cause impacts to the users, so the users should accept or be notified before these changes are effective.

However, there is currently no constraint to prevent the authorities from modifying these variables without notifying the users.

The controllable privileged state update functions are as follows:

| Target | Function | Modifier |
|---|---|---|
| Coin98DollarMintBurn.sol (L:836) | setLimitTime() | onlyOwner |
| Coin98DollarMintBurn.sol (L:848) | setExchangeFee() | onlyOwner |
| Coin98DollarMintBurn.sol (L:864) | setExchangeFeeBurner() | onlyOwner |
| Coin98DollarMintBurn.sol (L:879) | setMinter() | onlyOwner |
| Coin98DollarMintBurn.sol (L:963) | setBurner() | onlyOwner |
| Coin98DollarMintBurn.sol (L:1017) | setMinterSupply() | onlyOwner |

| | | |
|---|---|---|
| Coin98DollarMintBurn.sol (L:1037) | setBurnerSupply() | onlyOwner |
| Coin98DollarMintBurn.sol (L:1232) | withdrawMultiple() | onlyOwner |

### 5.2.2. Remediation

In the ideal case, the critical state variables should not be modifiable to keep the integrity of the smart contract. However, if modifications are needed, Inspex suggests implementing a community-run smart contract governance to control the use of these functions.

If removing the functions or implementing the smart contract governance is not possible, Inspex suggests mitigating the risk of this issue by using a timelock mechanism to delay the changes for a reasonable amount of time, e.g., 24 hours.

## 5.3. Incorrect Daily Total Mint and Burn Amount Update

| | |
|---|---|
| **ID** | IDX-003 |
| **Target** | Coin98DollarMintBurn |
| **Category** | Advanced Smart Contract Vulnerability |
| **CWE** | CWE-840: Business Logic Errors |
| **Risk** | **Severity:** Medium<br><br>**Impact:** Low<br>When the time has passed for `LIMIT_TIME` hours (24 hours by default) since the first mint / burn of the day, the `currentTotalMintedToday` or `currentTotalBurnedToday` value does not reset to zero due to the incorrect total mint / burn amount updates. This could prevent users from minting / burning $cUSD for the expected amount.<br><br>**Likelihood:** High<br>It is very likely that the total mint and burn amount per `LIMIT_TIME` will be incorrectly updated every time. |
| **Status** | Resolved<br>The Coin98 team has resolved this issue as suggested by updating the `minter.lastExchange` and `burner.lastExchange` values only when the `diffHours` is greater than or equal to the `LIMIT_TIME` in commit `40d70294a12bbc9692fc47195a3a5bc488b9f89f`. |

### 5.3.1. Description

The `Coin98DollarMintBurn` contract is used to mint / burn $cUSD from the value of provided asset tokens, with a limited mint / burn quantity of $cUSD in `LIMIT_TIME` hours.

The mint / burn `totalSupplyPerDay` of the $cUSD are validated in the `checkTotalMinted()` and `checkTotalBurned()` functions. When users mint / burn $cUSD, it calls the `checkTotalMinted()` or `checkTotalBurned()` function to update the `minter.lastExchange` value in line 1078 and the `burner.lastExchange` in line 1104. The `lastExchange` value is used to compare with the next `block.timestamp` when users mint/burn for the next time to check if the time has already passed for `LIMIT_TIME` hours.

But the `lastExchange` is also updated when `diffHours` is less than the `LIMIT_TIME`, which causes the `currentTotalMintedToday` and `currentTotalBurnedToday` to take longer to reset.

For example, the `minter.totalSupplyPerDay` value is 100. The user minted 50 $cUSD today at 2 a.m., thus the `currentTotalMintedToday` value is 50 and the `minter.lastExchange` value is updated. The next day, at 1 a.m., the user mints 40 $cUSD, resulting in the `currentTotalMintedToday` value of 90, and the `minter.lastExchange` value is updated once more. Even though it has been 24 (`LIMIT_TIME`) hours from

the first mint, a user who wishes to mint for more than 10 $cUSD on the same day at 3 a.m. is unable to do so. Because `block.timestamp - minter.lastExchange` is still less than `LIMIT_TIME` so the `currentTotalMintedToday` is not reset.

**Coin98DollarMintBurn.sol**

```
1057 function checkTotalMinted(TokenMinter storage minter, uint256 amount)
1058     internal
1059     returns (uint256)
1060 {
1061     uint256 diffHours = (block.timestamp - minter.lastExchange) / 60 / 60;
1062 
1063     uint256 currentTotalMinted = minter.totalMinted.add(amount);
1064     uint256 currentTotalMintedToday = diffHours >= LIMIT_TIME
1065         ? 0
1066         : minter.totalPerDay;
1067     currentTotalMintedToday = currentTotalMintedToday.add(amount);
1068 
1069     require(
1070     currentTotalMintedToday <= minter.totalSupplyPerDay &&
1071         currentTotalMinted <= minter.totalSupply,
1072     "Coin98DollarMintBurn: Amount must be less than total supply and total per
     day"
1073     );
1074 
1075     // Update tracking information
1076     minter.totalMinted = currentTotalMinted;
1077     minter.totalPerDay = currentTotalMintedToday;
1078     minter.lastExchange = block.timestamp;
1079     return currentTotalMintedToday;
1080 }
1081 
1082 /// @notice Check total supply and total supply per day of burner
1083 function checkTotalBurned(TokenBurner storage burner, uint256 amount)
1084     internal
1085     returns (uint256)
1086 {
1087     uint256 diffHours = (block.timestamp - burner.lastExchange) / 60 / 60;
1088 
1089     uint256 currentTotalBurned = burner.totalBurned.add(amount);
1090     uint256 currentTotalBurnedToday = diffHours >= LIMIT_TIME
1091         ? 0
1092         : burner.totalPerDay;
1093     currentTotalBurnedToday = currentTotalBurnedToday.add(amount);
1094 
1095     require(
1096         currentTotalBurnedToday <= burner.totalSupplyPerDay &&
1097             currentTotalBurned <= burner.totalSupply,
```

```
1098          "Coin98DollarMintBurn: Amount must be less than total supply and total
      per day"
1099          );
1100
1101      // Update tracking information
1102      burner.totalBurned = currentTotalBurned;
1103      burner.totalPerDay = currentTotalBurnedToday;
1104      burner.lastExchange = block.timestamp;
1105      return currentTotalBurnedToday;
1106  }
```

### 5.3.2. Remediation

Inspex suggests updating the `minter.lastExchange` and `burner.lastExchange` values only when the `diffHours` is greater than or equal to the `LIMIT_TIME` in lines 1066 and 1094.

**Coin98DollarMintBurn.sol**

```
1057  function checkTotalMinted(TokenMinter storage minter, uint256 amount)
1058      internal
1059      returns (uint256)
1060  {
1061      uint256 diffHours = (block.timestamp - minter.lastExchange) / 60 / 60;
1062
1063      uint256 currentTotalMinted = minter.totalMinted.add(amount);
1064      if (diffHours >= LIMIT_TIME) {
1065          currentTotalMintedToday = 0;
1066          minter.lastExchange = block.timestamp;
1067      } else {
1068          currentTotalMintedToday = minter.totalPerDay;
1069      }
1070      currentTotalMintedToday = currentTotalMintedToday.add(amount);
1071
1072      require(
1073      currentTotalMintedToday <= minter.totalSupplyPerDay &&
1074          currentTotalMinted <= minter.totalSupply,
1075      "Coin98DollarMintBurn: Amount must be less than total supply and total per
      day"
1076      );
1077
1078      // Update tracking information
1079      minter.totalMinted = currentTotalMinted;
1080      minter.totalPerDay = currentTotalMintedToday;
1081      return currentTotalMintedToday;
1082  }
1083
1084  /// @notice Check total supply and total supply per day of burner
1085  function checkTotalBurned(TokenBurner storage burner, uint256 amount)
```

```
1086      internal
1087      returns (uint256)
1088  {
1089      uint256 diffHours = (block.timestamp - burner.lastExchange) / 60 / 60;
1090
1091      uint256 currentTotalBurned = burner.totalBurned.add(amount);
1092      if (diffHours >= LIMIT_TIME) {
1093          currentTotalBurnedToday = 0;
1094          burner.lastExchange = block.timestamp;
1095      } else {
1096          currentTotalBurnedToday = burner.totalPerDay;
1097      }
1098      currentTotalBurnedToday = currentTotalBurnedToday.add(amount);
1099
1100      require(
1101          currentTotalBurnedToday <= burner.totalSupplyPerDay &&
1102              currentTotalBurned <= burner.totalSupply,
1103          "Coin98DollarMintBurn: Amount must be less than total supply and total
      per day"
1104          );
1105
1106      // Update tracking information
1107      burner.totalBurned = currentTotalBurned;
1108      burner.totalPerDay = currentTotalBurnedToday;
1109      return currentTotalBurnedToday;
1110  }
```

## 5.4. Division Before Multiplication

| | |
|---|---|
| **ID** | IDX-004 |
| **Target** | Coin98DollarMintBurn |
| **Category** | Advanced Smart Contract Vulnerability |
| **CWE** | CWE-840: Business Logic Errors |
| **Risk** | **Severity:** Low<br><br>**Impact:** Low<br>The rounding error can cause the `amountToBurn` to be slightly miscalculated.<br><br>**Likelihood:** Medium<br>It is likely that the result from multiplying the `amount` with the `minter.percents` then dividing by `Percent` can result in a decimal value. |
| **Status** | Resolved<br>The Coin98 team has resolved this issue as suggested by modifying the `mint()` function to perform multiplication before division in commit `40d70294a12bbc9692fc47195a3a5bc488b9f89f`. |

### 5.4.1. Description

Solidity supports only integer values but not floating point values. The division of integers can result in a value with decimal points, which will be rounded off. This rounding error can cause the calculation to be different from what it should be, especially when that value is later multiplied with another value

In line 1190, the `amountToBurn` value is calculated by multiplying the `valueByPercent` (which is a division result in line: 1181) by `10**tokenDecimals`, and then doing the rest of the math operation. The rounding error caused by the division is amplified in the multiplication and can increase the amount of miscalculation.

**Coin98DollarMintBurn.sol**

```
1168 function mint(uint256 _id, uint256 amount) public onlyActiveMinter(_id) {
1169     require(
1170         amount > 0,
1171         "Coin98DollarMintBurn: Amount must be a positive number and greater
     than zero"
1172     );
1173     TokenMinter storage minter = TokenMinters[_id];
1174
1175     uint256 currentTotalMintedToday = checkTotalMinted(minter, amount);
1176
1177     uint256[] memory amountToTransfer = new uint256[](minter.pairs.length);
1178
```

```
1179        for (uint256 i = 0; i < minter.pairs.length; i++) {
1180            uint256 tokenDecimals = minter.decimals[i];
1181            uint256 valueByPercent = amount.mul(minter.percents[i]).div(
1182                Percent
1183            );
1184
1185            // Feed the latest price by ChainLink
1186            (uint256 price, uint256 priceDecimals) = getLatestPrice(
1187                minter.priceFeed[i]
1188            );
1189
1190            uint256 amountToBurn = valueByPercent
1191                .mul(10**tokenDecimals)
1192                .div(price)
1193                .mul(10**priceDecimals)
1194                .div(BASE_DECIMALS);
1195
1196            // Transfer money first before do anything effects
1197            IERC20(minter.pairs[i]).safeTransferFrom(
1198                msg.sender,
1199                address(this),
1200                amountToBurn
1201            );
1202            amountToTransfer[i] = amountToBurn;
1203        }
1204
1205        uint256 systemFee = amount.mul(minter.systemFee).div(Percent);
1206        // Update tracking information
1207         minter.totalSystemFee = minter.totalSystemFee.add(systemFee);
1208
1209        // Claim system fee for each exchange cUSD
1210        amount = amount.sub(systemFee);
1211
1212        require(
1213            amount > 0,
1214            "Coin98DollarMintBurn: Total Mint must be a positive number and greater
    than zero"
1215        );
1216
1217        // Mint CUSD Token to .sender
1218        CUSD_TOKEN.mint(msg.sender, amount);
1219
1220        emit Mint(
1221            _id,
1222            msg.sender,
1223            amountToTransfer,
1224            amount,
```

```
1225            systemFee,
1226            currentTotalMintedToday
1227        );
1228    }
```

### 5.4.2. Remediation

Inspex suggests modifying the affected lines of code to perform multiplication before division, for example:

**Coin98DollarMintBurn.sol**

```
1168    function mint(uint256 _id, uint256 amount) public onlyActiveMinter(_id) {
1169        require(
1170            amount > 0,
1171            "Coin98DollarMintBurn: Amount must be a positive number and greater
        than zero"
1172        );
1173        TokenMinter storage minter = TokenMinters[_id];
1174
1175        uint256 currentTotalMintedToday = checkTotalMinted(minter, amount);
1176
1177        uint256[] memory amountToTransfer = new uint256[](minter.pairs.length);
1178
1179        for (uint256 i = 0; i < minter.pairs.length; i++) {
1180            uint256 tokenDecimals = minter.decimals[i];
1181
1182            // Feed the latest price by ChainLink
1183            (uint256 price, uint256 priceDecimals) = getLatestPrice(
1184                minter.priceFeed[i]
1185            );
1186
1187            uint256 amountToBurn = amount
1188                .mul(minter.percents[i])
1189                .mul(10**tokenDecimals)
1190                .mul(10**priceDecimals)
1191                .div(Percent)
1192                .div(price)
1193                .div(BASE_DECIMALS);
1194
1195            // Transfer money first before do anything effects
1196            IERC20(minter.pairs[i]).safeTransferFrom(
1197                msg.sender,
1198                address(this),
1199                amountToBurn
1200            );
1201            amountToTransfer[i] = amountToBurn;
1202        }
1203
```

```
1204        uint256 systemFee = amount.mul(minter.systemFee).div(Percent);
1205        // Update tracking information
1206         minter.totalSystemFee = minter.totalSystemFee.add(systemFee);
1207
1208        // Claim system fee for each exchange cUSD
1209        amount = amount.sub(systemFee);
1210
1211        require(
1212            amount > 0,
1213            "Coin98DollarMintBurn: Total Mint must be a positive number and greater
    than zero"
1214        );
1215
1216        // Mint CUSD Token to .sender
1217        CUSD_TOKEN.mint(msg.sender, amount);
1218
1219        emit Mint(
1220            _id,
1221            msg.sender,
1222            amountToTransfer,
1223            amount,
1224            systemFee,
1225            currentTotalMintedToday
1226        );
1227    }
```

## 5.5. Insufficient Logging for Privileged Functions

| | |
|---|---|
| **ID** | IDX-005 |
| **Target** | Coin98DollarMintBurn |
| **Category** | General Smart Contract Vulnerability |
| **CWE** | CWE-778: Insufficient Logging |
| **Risk** | **Severity:** Very Low<br><br>**Impact:** Low<br>Privileged functions' executions cannot be monitored easily by the users.<br><br>**Likelihood:** Low<br>It is not likely that the execution of the privileged functions will be a malicious action. |
| **Status** | Resolved<br>The Coin98 team has resolved this issue as suggested by emitting events for the execution of privileged functions in commit `40d70294a12bbc9692fc47195a3a5bc488b9f89f`. |

### 5.5.1. Description

Privileged functions that are executable by the controlling parties are not logged properly by emitting events. Without events, it is not easy for the public to monitor the execution of those privileged functions, allowing the controlling parties to perform actions that cause big impacts on the platform.

The owner could withdraw token by executing the `withdrawMultiple()` function in the `Coin98DollarMintBurn` contract, and no events are emitted, for example:

**Coin98DollarMintBurn.sol**

```
1232 function withdrawMultiple(address[] calldata tokens) public onlyOwner {
1233     for (uint256 i = 0; i < tokens.length; i++) {
1234         if (tokens[i] == address(0)) {
1235             payable(msg.sender).transfer(address(this).balance);
1236         } else {
1237             IERC20 token = IERC20(tokens[i]);
1238 
1239             uint256 tokenBalance = token.balanceOf(address(this));
1240             if (tokenBalance > 0) {
1241                 token.safeTransfer(msg.sender, tokenBalance);
1242             }
1243         }
1244     }
1245 }
```

### 5.5.2. Remediation

Inspex suggests emitting events for the execution of privileged functions, for example:

**Coin98DollarMintBurn.sol**

```
1232 event WithdrawToken(address[] calldata tokens);
1233 function withdrawMultiple(address[] calldata tokens) public onlyOwner {
1234     for (uint256 i = 0; i < tokens.length; i++) {
1235         if (tokens[i] == address(0)) {
1236             payable(msg.sender).transfer(address(this).balance);
1237         } else {
1238             IERC20 token = IERC20(tokens[i]);
1239 
1240             uint256 tokenBalance = token.balanceOf(address(this));
1241             if (tokenBalance > 0) {
1242                 token.safeTransfer(msg.sender, tokenBalance);
1243             }
1244         }
1245     }
1246     emit WithdrawToken(tokens);
1247 }
```

## 5.6. Improper Function Visibility

| | |
|---|---|
| **ID** | IDX-006 |
| **Target** | Coin98DollarMintBurn |
| **Category** | Smart Contract Best Practice |
| **CWE** | CWE-710: Improper Adherence to Coding Standards |
| **Risk** | **Severity:** Info<br>**Impact: None**<br>**Likelihood: None** |
| **Status** | Resolved<br>The Coin98 team has resolved this issue as suggested by changing all functions' visibility to `external` if they are not called from any functions in the contract at commit `40d70294a12bbc9692fc47195a3a5bc488b9f89f`. |

### 5.6.1. Description

Public functions that are never called internally by the contract itself should have external visibility. This improves the readability of the contract, allowing clear distinction between functions that are externally used and functions that are also called internally.

The following source code shows that the `setLimitTime()` function on the `Coin98DollarMintBurn` contact is set to public and it is never called from any internal function.

**Coin98DollarMintBurn.sol**

```
836 function setLimitTime(uint256 _limitTime) public onlyOwner {
837     require(
838         _limitTime > 0,
839         "Coin98DollarMintBurn: Limit time must be a positive number and greater
    than zero"
840     );
841     LIMIT_TIME = _limitTime;
842
843     emit UpdateLimitTime(_limitTime);
844 }
```

The following table contains all functions that have public visibility and never be called from any internal function.

| File | Contract | Function |
|---|---|---|
| Coin98DollarMintBurn.sol (L:836) | Coin98DollarMintBurn | setLimitTime() |
| Coin98DollarMintBurn.sol (L:848) | Coin98DollarMintBurn | setExchangeFee() |
| Coin98DollarMintBurn.sol (L:864) | Coin98DollarMintBurn | setExchangeFeeBurner() |
| Coin98DollarMintBurn.sol (L:879) | Coin98DollarMintBurn | setMinter() |
| Coin98DollarMintBurn.sol (L:963) | Coin98DollarMintBurn | setBurner() |
| Coin98DollarMintBurn.sol (L:1017) | Coin98DollarMintBurn | setMinterSupply() |
| Coin98DollarMintBurn.sol (L:1037) | Coin98DollarMintBurn | setBurnerSupply() |
| Coin98DollarMintBurn.sol (L:1112) | Coin98DollarMintBurn | burn() |
| Coin98DollarMintBurn.sol (L:1232) | Coin98DollarMintBurn | withdrawMultiple() |

### 5.6.2. Remediation

Inspex suggests changing all functions' visibility to external if they are not called from any internal function, as shown in the following example:

**Coin98DollarMintBurn.sol**

```
836 function setLimitTime(uint256 _limitTime) external onlyOwner {
837     require(
838         _limitTime > 0,
839         "Coin98DollarMintBurn: Limit time must be a positive number and greater
    than zero"
840     );
841     LIMIT_TIME = _limitTime;
842
843     emit UpdateLimitTime(_limitTime);
844 }
```

# 6. Appendix

## 6.1. About Inspex


Inspex is formed by a team of cybersecurity experts highly experienced in various fields of cybersecurity. We provide blockchain and smart contract professional services at the highest quality to enhance the security of our clients and the overall blockchain ecosystem.

**Follow Us On:**

| | |
|---|---|
| **Website** | https://inspex.co |
| **Twitter** | @InspexCo |
| **Facebook** | https://www.facebook.com/InspexCo |
| **Telegram** | @inspex_announcement |

inspeX

CYBERSECURITY PROFESSIONAL SERVICE