Solana DollarMintBurn

Smart Contract Audit Report
Prepared for Coin98

Date Issued:	Jul 26, 2022
Project ID:	AUDIT2022042
Version:	v0.2
Confidentiality Level:	Confidential

inspeX
CYBERSECURITY PROFESSIONAL SERVICE

Confidential

Report Information

Project ID	AUDIT2022042
Version	v0.2
Client	Coin98
Project	Solana DollarMintBurn
Auditor(s)	Ronnachai Chaipha Darunphop Pengkumta Sorawish Laovakul
Author(s)	Sorawish Laovakul
Reviewer	Natsasit Jirathammanuwat
Confidentiality Level	Confidential

Version History

Version	Date	Description	Author(s)
0.1	Jul 7, 2022	Draft report	Sorawish Laovakul
0.2	Jul 26, 2022	Add issue IDX-002	Ronnachai Chaipha

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 Solana DollarMintBurn smart contracts between Jun 27, 2022 and Jun 29, 2022. During the audit, Inspex team examined all smart contracts and the overall operation within the scope to understand the overview of Solana DollarMintBurn 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 1 critical, 4 high, 2 medium, 1 low, and 1 very low-severity issues. With the project team's prompt response, 1 critical, 3 high, 2 medium, 1 low, and 1 very low-severity issues were resolved or mitigated in the reassessment, while 1 high-severity issues were acknowledged by the team. 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.

Solana DollarMintBurn is the contract on the Solana chain that provides the mint and burn cUSD token mechanism to users. The users can mint the cUSD token by transferring the tokens to the contract as a collateral. The asset tokens will be transferred back when users burn the cUSD token.

Scope Information:

Project Name	Solana DollarMintBurn
Website	https://coin98.com/
Smart Contract Type	Solana Program
Chain	Solana
Programming Language	Rust
Category	Token, Stable Coin

Audit Information:

Audit Method	Whitebox
Audit Date	Jun 27, 2022 - Jun 29, 2022
Reassessment Date	Jul 6, 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: -)

Contract	Location (URL)
coin98_dollar_mint_burn	-

Reassessment: (Commit: -)

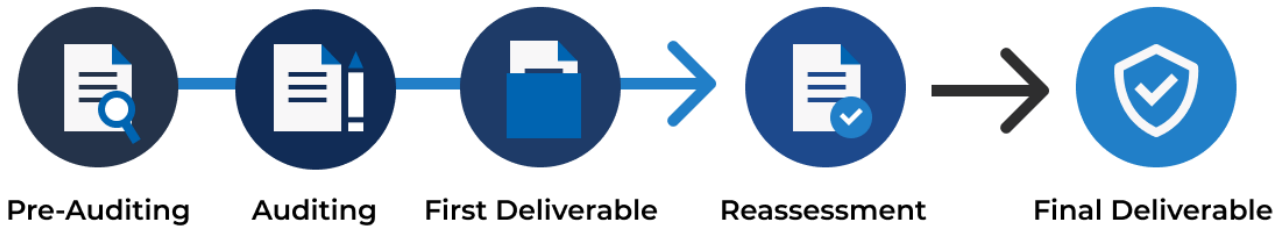
Contract	Location (URL)
coin98_dollar_mint_burn	-

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 1.2. The latest stable compiler version should be used 1.3. The circuit breaker mechanism should not prevent users from withdrawing their funds 1.4. The smart contract source code should be publicly available 1.5. State variables should not be unfairly controlled by privileged accounts 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 2.2. Contract ownership should not be modifiable by unauthorized actors 2.3. Access control should be defined and enforced for each actor roles 2.4. Authentication measures must be able to correctly identify the user 2.5. Smart contract initialization should be done only once by an authorized party 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 3.2. Privileged functions or modifications of critical states should be logged 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 4.2. Measures should be implemented to prevent undesired effects from the ordering of transactions 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 5.2. Spot price should not be used as a data source for price oracles 5.3. Timestamp should not be used to execute critical functions 5.4. Plain sensitive data should not be stored on-chain 5.5. Modification of array state should not be done by value 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 6.2. Funds should not be approved or transferred to unknown accounts 6.3. Reentrant calling should not negatively affect the contract states 6.4. Vulnerable or outdated components should not be used in the smart contract 6.5. Deprecated components that have no longer been supported should not be used in the smart contract 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 7.2. Explicit conversion of types should be checked to prevent unexpected results 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 8.2. Unexpected revert should not make the whole smart contract unusable 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 9.2. Token implementation should comply with the standard specification 9.3. Floating pragma version should not be used 9.4. Builtin symbols should not be shadowed 9.5. Functions that are never called internally should not have public visibility 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**.

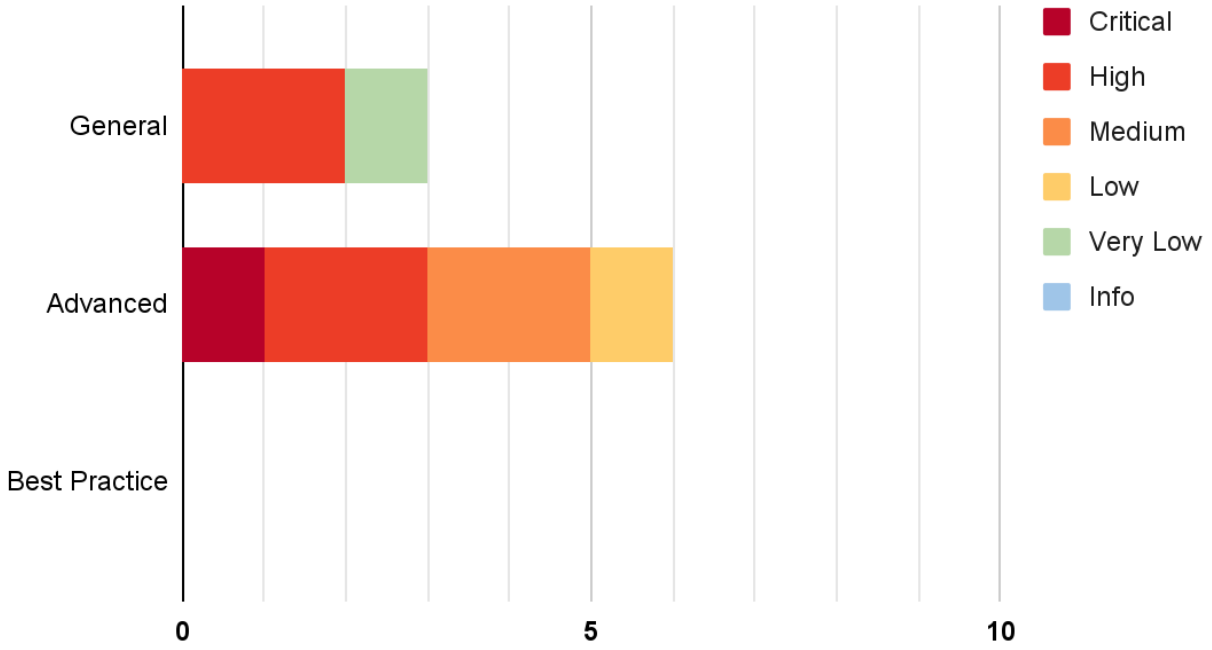
Severity is the overall risk of the issue. It can be categorized into five levels: **Very Low**, **Low**, **Medium**, **High**, and **Critical**. It is calculated from the combination of likelihood and impact factors using the matrix below. The severity of findings with no likelihood or impact would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

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

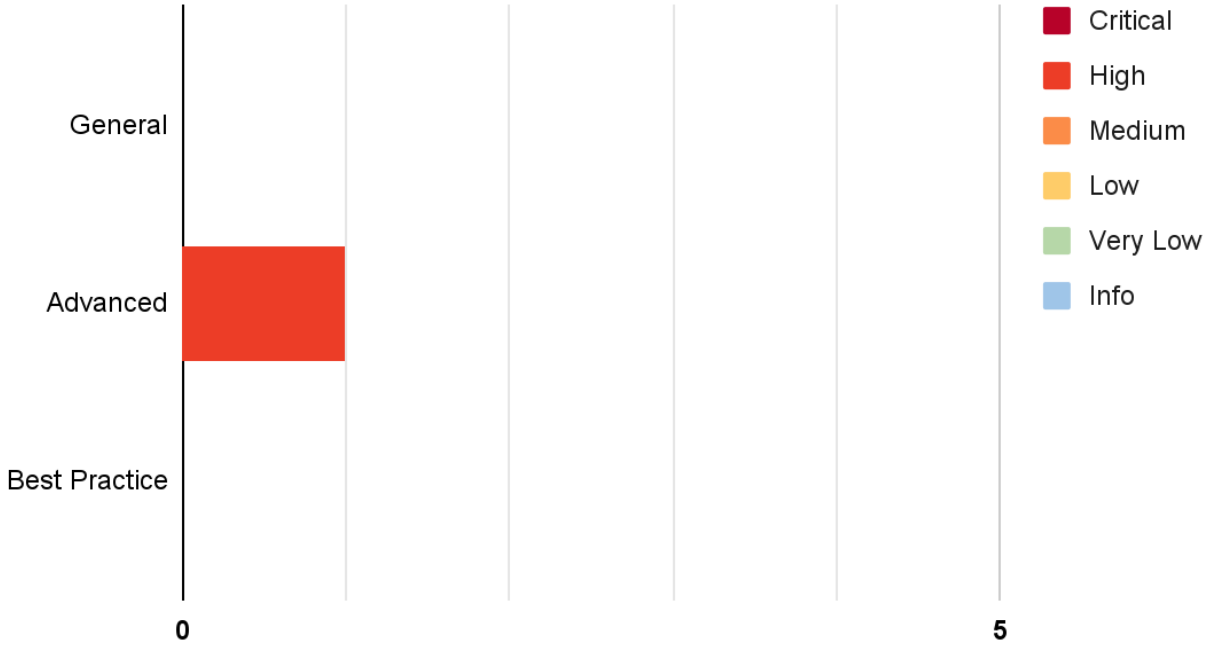
4. Summary of Findings

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	Lack of price_feed Account Validation in burn() Function	Advanced	Critical	Resolved
IDX-002	Lack of Token Decimal Conversion	Advanced	High	Resolved
IDX-003	Upgradability of Solana Program	General	High	Resolved *
IDX-004	Centralized Control of State Variable	General	High	Resolved *
IDX-005	Design Flaw in cUSD Token	Advanced	High	Acknowledged
IDX-006	Incorrect Logic Operator	Advanced	Medium	Resolved
IDX-007	Incorrect Update Account State	Advanced	Medium	Resolved
IDX-008	Unbound Configuration Parameter	Advanced	Low	Resolved
IDX-009	Insufficient Logging for Privileged Functions	General	Very Low	Resolved

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

5. Detailed Findings Information

5.1. Lack of price_feed Account Validation in burn() Function

ID	IDX-001
Target	coin98_dollar_mint_burn
Category	Advanced Smart Contract Vulnerability
CWE	CWE-20: Improper Input Validation
Risk	**Severity:** Critical **Impact:** High The price rate that is used to calculate the output token amount can be manipulated by inputting any token's price feed account. As a result, an attacker can use another price feed to drain the asset token that has a higher value than the using price feed pair. **Likelihood:** High The platform users can input any price feed account to determine the asset token's price without any restrictions.
Status	Resolved The Coin98 team has resolved this issue by implementing the validation mechanism to check if the `price_feed` account is the expected account. This issue has been resolved in commit `9ad82315616fe6b71dc9bbc62f2c2fba194727e9`.

5.1.1. Description

The burning procedure in the `coin98_dollar_mint_burn` program allows users to burn the cUSD token in order to obtain the asset token.

Users must also supply the ChainLink price feed account to the program so that the program will use it for fetching the asset token's price, which is used to calculate the output token amount, at the following source code in lines 275-280.

lib.rs

```
248 pub fn burn<'a>(
249   ctx: Context<'_, '_, '_, 'a, BurnContext<'a>>,
250   amount: u64,
251 ) -> Result<()> {
252
253   let user = &ctx.accounts.user;
254   let app_data = &ctx.accounts.app_data;
255   let burner = &ctx.accounts.burner;
256
```

```
257   if !burner.is_active {
258     return Err(ErrorCode::Unavailable.into());
259   }
260
261   let current_timestamp = Clock::get().unwrap().unix_timestamp;
262   let timestamp_per_period = i64::from(app_data.limit) * 3600;
263   let is_in_period = burner.last_period_timestamp + timestamp_per_period <
      current_timestamp;
264   let current_period_burned_amount = if is_in_period {
      burner.per_period_burned_amount } else { 0u64 };
265
266   if current_period_burned_amount + amount > burner.per_period_burned_limit {
267     return Err(ErrorCode::LimitReached.into());
268   }
269   if burner.total_burned_amount + amount > burner.total_burned_limit {
270     return Err(ErrorCode::LimitReached.into());
271   }
272
273   let chainlink_program = &ctx.accounts.chainlink_program;
274   let accounts = &ctx.remaining_accounts;
275   let price_feed = &accounts[0];
276   let (price, precision) = get_price_feed(
277       &*chainlink_program,
278       &*price_feed,
279     );
280   let output_amount = multiply_fraction(amount, precision, price);
281
282   let pool_cusd = &ctx.accounts.pool_cusd;
283   let user_cusd = &ctx.accounts.user_cusd;
284   transfer_token(
285       &*user,
286       &user_cusd.to_account_info(),
287       &pool_cusd.to_account_info(),
288       amount,
289       &[],
290     )
291     .expect("CUSD Factory: CPI failed.");
292
293   let root_signer = &ctx.accounts.root_signer;
294   let cusd_mint = &ctx.accounts.cusd_mint;
295   let seeds: &[&[u8]] = &[
296     ROOT_SIGNER_SEED_1,
297     ROOT_SIGNER_SEED_2,
298     &[app_data.signer_nonce],
299   ];
300   burn_token(
301       &*root_signer,
```

```
302         &*cusd_mint,
303         &pool_cusd.to_account_info(),
304         amount,
305         &[&seeds],
306       )
307       .expect("CUSD Factory: CPI failed.");
308
309     let burner = &mut ctx.accounts.burner;
310     burner.total_burned_amount = burner.total_burned_amount + amount;
311     burner.per_period_burned_limit = current_period_burned_amount + amount;
312     if !is_in_period {
313       burner.last_period_timestamp = current_timestamp;
314     }
315     let protocol_fee = multiply_fraction(output_amount,
    u64::from(burner.fee_percent), 10000);
316     let amount_to_transfer = output_amount.checked_sub(protocol_fee).unwrap();
317     burner.accumulated_fee =
    burner.accumulated_fee.checked_add(protocol_fee).unwrap();
318
319     let pool_token = &accounts[1];
320     let pool_token =
    TokenAccount::unpack_from_slice(&pool_token.try_borrow_data().unwrap()).unwrap(
    );
321     if pool_token.owner != root_signer.key() || pool_token.mint !=
    burner.output_token {
322       return Err(ErrorCode::InvalidAccount.into());
323     }
324     let user_token = &accounts[2];
325     let user_token =
    TokenAccount::unpack_from_slice(&user_token.try_borrow_data().unwrap()).unwrap(
    );
326     if user_token.mint != burner.output_token {
327       return Err(ErrorCode::InvalidAccount.into());
328     }
329     transfer_token(
330         &*root_signer,
331         &accounts[1],
332         &accounts[2],
333         amount_to_transfer,
334         &[&seeds],
335       )
336       .expect("CUSD Factory: CPI failed.");
337
338     Ok(())
339 }
```

However, the `coin98_dollar_mint_burn` program has no validation for the ChainLink price feed account,

causing malicious users to submit another ChainLink price feed account. This results in the output token being miscalculated. The malicious user can receive more asset tokens than they should be.

5.1.2. Remediation

Inspex suggests implementing the validation mechanism to ensure the `price_feed` account is the expected account. For example, check that the `price_feed` account must be equal to the `output_price_feed` address in the `burner` account as shown in lines 276-278.

lib.rs

```
248 pub fn burn<'a>(
249   ctx: Context<'_, '_, '_, 'a, BurnContext<'a>>,
250   amount: u64,
251 ) -> Result<()> {
252
253   let user = &ctx.accounts.user;
254   let app_data = &ctx.accounts.app_data;
255   let burner = &ctx.accounts.burner;
256
257   if !burner.is_active {
258     return Err(ErrorCode::Unavailable.into());
259   }
260
261   let current_timestamp = Clock::get().unwrap().unix_timestamp;
262   let timestamp_per_period = i64::from(app_data.limit) * 3600;
263   let is_in_period = burner.last_period_timestamp + timestamp_per_period <
    current_timestamp;
264   let current_period_burned_amount = if is_in_period {
    burner.per_period_burned_amount } else { 0u64 };
265
266   if current_period_burned_amount + amount > burner.per_period_burned_limit {
267     return Err(ErrorCode::LimitReached.into());
268   }
269   if burner.total_burned_amount + amount > burner.total_burned_limit {
270     return Err(ErrorCode::LimitReached.into());
271   }
272
273   let chainlink_program = &ctx.accounts.chainlink_program;
274   let accounts = &ctx.remaining_accounts;
275   let price_feed = &accounts[0];
276   if price_feed.key() != burner.output_price_feed {
277     return Err(ErrorCode::InvalidAccount.into());
278   }
279   let (price, precision) = get_price_feed(
280       &*chainlink_program,
281       &*price_feed,
282     );
```

```
283   let output_amount = multiply_fraction(amount, precision, price);
284
285   let pool_cusd = &ctx.accounts.pool_cusd;
286   let user_cusd = &ctx.accounts.user_cusd;
287   transfer_token(
288       &*user,
289       &user_cusd.to_account_info(),
290       &pool_cusd.to_account_info(),
291       amount,
292       &[],
293     )
294     .expect("CUSD Factory: CPI failed.");
295
296   let root_signer = &ctx.accounts.root_signer;
297   let cusd_mint = &ctx.accounts.cusd_mint;
298   let seeds: &[&[u8]] = &[
299     ROOT_SIGNER_SEED_1,
300     ROOT_SIGNER_SEED_2,
301     &[app_data.signer_nonce],
302   ];
303   burn_token(
304       &*root_signer,
305       &*cusd_mint,
306       &pool_cusd.to_account_info(),
307       amount,
308       &[&seeds],
309     )
310     .expect("CUSD Factory: CPI failed.");
311
312   let burner = &mut ctx.accounts.burner;
313   burner.total_burned_amount = burner.total_burned_amount + amount;
314   burner.per_period_burned_limit = current_period_burned_amount + amount;
315   if !is_in_period {
316     burner.last_period_timestamp = current_timestamp;
317   }
318   let protocol_fee = multiply_fraction(output_amount,
    u64::from(burner.fee_percent), 10000);
319   let amount_to_transfer = output_amount.checked_sub(protocol_fee).unwrap();
320   burner.accumulated_fee =
    burner.accumulated_fee.checked_add(protocol_fee).unwrap();
321
322   let pool_token = &accounts[1];
323   let pool_token =
    TokenAccount::unpack_from_slice(&pool_token.try_borrow_data().unwrap()).unwrap(
    );
324   if pool_token.owner != root_signer.key() || pool_token.mint !=
    burner.output_token {
```

```
325       return Err(ErrorCode::InvalidAccount.into());
326     }
327     let user_token = &accounts[2];
328     let user_token =
    TokenAccount::unpack_from_slice(&user_token.try_borrow_data().unwrap()).unwrap(
329 );
    if user_token.mint != burner.output_token {
330       return Err(ErrorCode::InvalidAccount.into());
331     }
332     transfer_token(
333         &*root_signer,
334         &accounts[1],
335         &accounts[2],
336         amount_to_transfer,
337         &[&seeds],
338       )
339       .expect("CUSD Factory: CPI failed.");
340 
341     Ok(())
342 }
```

Please note that the remediation for other issues are not yet applied in the examples above.

5.2. Lack of Token Decimal Conversion

ID	IDX-002
Target	coin98_dollar_mint_burn
Category	Advanced Smart Contract Vulnerability
CWE	CWE-840: Business Logic Errors
Risk	**Severity:** High **Impact:** High The output token can be miscalculated by inputting a token whose decimal does not match all input and output tokens. It results in users receiving an output token worth more or less than the value of the input token. **Likelihood:** Medium This issue is under the control of the platform owner and occurs when the token with different decimals is set by the platform owner.
Status	Resolved The Coin98 team has resolved this issue by adding a token decimal conversion mechanism to convert between the input and output token and a minimum amount for minting or burning. However, the `burn()` function has changed. The `amount` parameter will be used as the output token instead of the cUSD token. This issue has been resolved in commit `70ed4a4461a0e2e8c4595dde8b4514ede86af939`.

5.2.1. Description

The `coin98_dollar_mint_burn` program allows users to mint and burn a token to produce an output token with the same value as the input token that is set in the minter and burner accounts using a price oracle.

The source code below shows the token input and output calculations without checking and calculating the token decimal between the input and output tokens. In the case that the tokens' decimals are different, the result will be miscalculated. As shown in lines 191 and 280.

lib.rs

```
142 pub fn mint<'a>(
143     ctx: Context<'_, '_, '_, 'a, MintContext<'a>>,
144     amount: u64,
145     extra_instructions: Vec<u8>,
146   ) -> Result<()> {
147 
148     let user = &ctx.accounts.user;
149     let app_data = &ctx.accounts.app_data;
```

```
150     let root_signer = &ctx.accounts.root_signer;
151     let minter = &ctx.accounts.minter;
152 
153     if !minter.is_active {
154       return Err(ErrorCode::Unavailable.into());
155     }
156 
157     let current_timestamp = Clock::get().unwrap().unix_timestamp;
158     let timestamp_per_period = i64::from(app_data.limit) * 3600;
159     let is_in_period = minter.last_period_timestamp + timestamp_per_period <
    current_timestamp;
160     let current_period_minted_amount = if is_in_period {
    minter.per_period_minted_amount } else { 0u64 };
161 
162     if current_period_minted_amount + amount > minter.per_period_minted_limit {
163       return Err(ErrorCode::LimitReached.into());
164     }
165     if minter.total_minted_amount + amount > minter.total_minted_limit {
166       return Err(ErrorCode::LimitReached.into());
167     }
168 
169     let chainlink_program = &ctx.accounts.chainlink_program;
170     let accounts = &ctx.remaining_accounts;
171 
172     let account_indices: Vec<usize> = extra_instructions.iter()
173       .map(|extra| {
174         usize::from(*extra)
175       })
176       .collect();
177 
178     for (i, input_token) in minter.input_tokens.iter().enumerate() {
179       let input_price_feed = &minter.input_price_feeds[i];
180       let price_feed = &accounts[3*i];
181       if price_feed.key() != *input_price_feed {
182         return Err(ErrorCode::InvalidAccount.into());
183       }
184       let (price, precision) = get_price_feed(
185           &*chainlink_program,
186           &*price_feed,
187         );
188       let value_contrib = minter.input_percentages[i];
189 
190       let input_vaule =
    amount.checked_mul(u64::from(value_contrib)).unwrap().checked_div(10000).unwrap
    ();
191       let input_amount = multiply_fraction(input_vaule, precision, price);
192 
```

```
193       let from_account_index = account_indices[3*i + 1];
194       let to_account_index = account_indices[3*i + 2];
195       let from_account = &accounts[from_account_index];
196       let from_account =
TokenAccount::unpack_from_slice(&from_account.try_borrow_data().unwrap()).unwra
p();
197       let to_account = &accounts[to_account_index];
198       let to_account =
TokenAccount::unpack_from_slice(&to_account.try_borrow_data().unwrap()).unwrap(
);
199       if from_account.mint != *input_token {
200         return Err(ErrorCode::InvalidAccount.into());
201       }
202       if to_account.mint != *input_token || to_account.owner !=
root_signer.key() {
203         return Err(ErrorCode::InvalidAccount.into());
204       }
205
206       transfer_token(
207           &*user,
208           &accounts[from_account_index],
209           &accounts[to_account_index],
210           input_amount,
211           &[],
212         )
213         .expect("CUSD Factory: CPI failed.");
214     }
215
216     let minter = &mut ctx.accounts.minter;
217     minter.total_minted_amount = minter.total_minted_amount + amount;
218     minter.per_period_minted_limit = current_period_minted_amount + amount;
219     if !is_in_period {
220       minter.last_period_timestamp = current_timestamp;
221     }
222
223     let protocol_fee = multiply_fraction(amount, u64::from(minter.fee_percent),
10000);
224     let amount_to_transfer = amount.checked_sub(protocol_fee).unwrap();
225     minter.accumulated_fee =
minter.accumulated_fee.checked_add(protocol_fee).unwrap();
226
227     let cusd_mint = &ctx.accounts.cusd_mint;
228     let recipient = &ctx.accounts.recipient;
229
230     let seeds: &[&[u8]] = &[
231       ROOT_SIGNER_SEED_1,
232       ROOT_SIGNER_SEED_2,
```

```
233         &[app_data.signer_nonce],
234       ];
235
236       mint_token(
237           &*root_signer,
238           &*cusd_mint,
239           &*recipient,
240           amount_to_transfer,
241           &[&seeds],
242         )
243         .expect("CUSD Factory: CPI failed.");
244
245       Ok(())
246   }
```

lib.rs

```
248 pub fn burn<'a>(
249     ctx: Context<'_, '_, '_, 'a, BurnContext<'a>>,
250     amount: u64,
251   ) -> Result<()> {
252
253     let user = &ctx.accounts.user;
254     let app_data = &ctx.accounts.app_data;
255     let burner = &ctx.accounts.burner;
256
257     if !burner.is_active {
258       return Err(ErrorCode::Unavailable.into());
259     }
260
261     let current_timestamp = Clock::get().unwrap().unix_timestamp;
262     let timestamp_per_period = i64::from(app_data.limit) * 3600;
263     let is_in_period = burner.last_period_timestamp + timestamp_per_period <
    current_timestamp;
264     let current_period_burned_amount = if is_in_period {
    burner.per_period_burned_amount } else { 0u64 };
265
266     if current_period_burned_amount + amount > burner.per_period_burned_limit {
267       return Err(ErrorCode::LimitReached.into());
268     }
269     if burner.total_burned_amount + amount > burner.total_burned_limit {
270       return Err(ErrorCode::LimitReached.into());
271     }
272
273     let chainlink_program = &ctx.accounts.chainlink_program;
274     let accounts = &ctx.remaining_accounts;
275     let price_feed = &accounts[0];
276     let (price, precision) = get_price_feed(
```

```
277         &*chainlink_program,
278         &*price_feed,
279       );
280     let output_amount = multiply_fraction(amount, precision, price);
281
282     let pool_cusd = &ctx.accounts.pool_cusd;
283     let user_cusd = &ctx.accounts.user_cusd;
284     transfer_token(
285         &*user,
286         &user_cusd.to_account_info(),
287         &pool_cusd.to_account_info(),
288         amount,
289         &[],
290       )
291       .expect("CUSD Factory: CPI failed.");
292
293     let root_signer = &ctx.accounts.root_signer;
294     let cusd_mint = &ctx.accounts.cusd_mint;
295     let seeds: &[&[u8]] = &[
296       ROOT_SIGNER_SEED_1,
297       ROOT_SIGNER_SEED_2,
298       &[app_data.signer_nonce],
299     ];
300     burn_token(
301         &*root_signer,
302         &*cusd_mint,
303         &pool_cusd.to_account_info(),
304         amount,
305         &[&seeds],
306       )
307       .expect("CUSD Factory: CPI failed.");
308
309     let burner = &mut ctx.accounts.burner;
310     burner.total_burned_amount = burner.total_burned_amount + amount;
311     burner.per_period_burned_limit = current_period_burned_amount + amount;
312     if !is_in_period {
313       burner.last_period_timestamp = current_timestamp;
314     }
315     let protocol_fee = multiply_fraction(output_amount,
    u64::from(burner.fee_percent), 10000);
316     let amount_to_transfer = output_amount.checked_sub(protocol_fee).unwrap();
317     burner.accumulated_fee =
    burner.accumulated_fee.checked_add(protocol_fee).unwrap();
318
319     let pool_token = &accounts[1];
320     let pool_token =
    TokenAccount::unpack_from_slice(&pool_token.try_borrow_data().unwrap()).unwrap(
```

```
    );
321     if pool_token.owner != root_signer.key() || pool_token.mint !=
    burner.output_token {
322       return Err(ErrorCode::InvalidAccount.into());
323     }
324     let user_token = &accounts[2];
325     let user_token =
    TokenAccount::unpack_from_slice(&user_token.try_borrow_data().unwrap()).unwrap(
    );
326     if user_token.mint != burner.output_token {
327       return Err(ErrorCode::InvalidAccount.into());
328     }
329     transfer_token(
330         &*root_signer,
331         &accounts[1],
332         &accounts[2],
333         amount_to_transfer,
334         &[&seeds],
335       )
336       .expect("CUSD Factory: CPI failed.");
337
338     Ok(())
339 }
```

5.2.2. Remediation

Inspex suggests applying the decimal calculation to properly converse between input and output tokens.

For example, in the `mint()` function at lines 192-193.

lib.rs

```
142 pub fn mint<'a>(
143     ctx: Context<'_, '_, '_, 'a, MintContext<'a>>,
144     amount: u64,
145     extra_instructions: Vec<u8>,
146   ) -> Result<()> {
147
148     let user = &ctx.accounts.user;
149     let app_data = &ctx.accounts.app_data;
150     let root_signer = &ctx.accounts.root_signer;
151     let minter = &ctx.accounts.minter;
152
153     if !minter.is_active {
154       return Err(ErrorCode::Unavailable.into());
155     }
156
157     let current_timestamp = Clock::get().unwrap().unix_timestamp;
```

```
158     let timestamp_per_period = i64::from(app_data.limit) * 3600;
159     let is_in_period = minter.last_period_timestamp + timestamp_per_period <
current_timestamp;
160     let current_period_minted_amount = if is_in_period {
minter.per_period_minted_amount } else { 0u64 };
161
162     if current_period_minted_amount + amount > minter.per_period_minted_limit {
163       return Err(ErrorCode::LimitReached.into());
164     }
165     if minter.total_minted_amount + amount > minter.total_minted_limit {
166       return Err(ErrorCode::LimitReached.into());
167     }
168
169     let chainlink_program = &ctx.accounts.chainlink_program;
170     let accounts = &ctx.remaining_accounts;
171
172     let account_indices: Vec<usize> = extra_instructions.iter()
173       .map(|extra| {
174         usize::from(*extra)
175       })
176       .collect();
177
178     for (i, input_token) in minter.input_tokens.iter().enumerate() {
179       let input_price_feed = &minter.input_price_feeds[i];
180       let price_feed = &accounts[3*i];
181       if price_feed.key() != *input_price_feed {
182         return Err(ErrorCode::InvalidAccount.into());
183       }
184       let (price, precision) = get_price_feed(
185           &*chainlink_program,
186           &*price_feed,
187         );
188       let value_contrib = minter.input_percentages[i];
189
190       let input_vaule =
amount.checked_mul(u64::from(value_contrib)).unwrap().checked_div(10000).unwrap
();
191       let input_amount = multiply_fraction(input_vaule, precision, price);
192       let input_precision = u64::pow(10, u32::from(minter.input_decimals[i]));
193       let input_amount = multiply_fraction(input_amount, input_precision,
CUSD_PRECISION);
194
195       let from_account_index = account_indices[3*i + 1];
196       let to_account_index = account_indices[3*i + 2];
197       let from_account = &accounts[from_account_index];
198       let from_account =
TokenAccount::unpack_from_slice(&from_account.try_borrow_data().unwrap()).unwra
```

```
p();
199       let to_account = &accounts[to_account_index];
200       let to_account =
TokenAccount::unpack_from_slice(&to_account.try_borrow_data().unwrap()).unwrap(
);
201       if from_account.mint != *input_token {
202         return Err(ErrorCode::InvalidAccount.into());
203       }
204       if to_account.mint != *input_token || to_account.owner !=
root_signer.key() {
205         return Err(ErrorCode::InvalidAccount.into());
206       }
207
208       transfer_token(
209           &*user,
210           &accounts[from_account_index],
211           &accounts[to_account_index],
212           input_amount,
213           &[],
214         )
215         .expect("CUSD Factory: CPI failed.");
216     }
217
218     let minter = &mut ctx.accounts.minter;
219     minter.total_minted_amount = minter.total_minted_amount + amount;
220     minter.per_period_minted_limit = current_period_minted_amount + amount;
221     if !is_in_period {
222       minter.last_period_timestamp = current_timestamp;
223     }
224
225     let protocol_fee = multiply_fraction(amount, u64::from(minter.fee_percent),
10000);
226     let amount_to_transfer = amount.checked_sub(protocol_fee).unwrap();
     minter.accumulated_fee =
227 minter.accumulated_fee.checked_add(protocol_fee).unwrap();
228
229     let cusd_mint = &ctx.accounts.cusd_mint;
230     let recipient = &ctx.accounts.recipient;
231
232     let seeds: &[&[u8]] = &[
233       ROOT_SIGNER_SEED_1,
234       ROOT_SIGNER_SEED_2,
235       &[app_data.signer_nonce],
236     ];
237
238     mint_token(
239         &*root_signer,
```

```
240           &*cusd_mint,
241           &*recipient,
242           amount_to_transfer,
243           &[&seeds],
244         )
245         .expect("CUSD Factory: CPI failed.");
246 
247     Ok(())
248 }
```

In the `burn()` function at the lines 281-282.

lib.rs

```
248 pub fn burn<'a>(
249     ctx: Context<'_, '_, '_, 'a, BurnContext<'a>>,
250     amount: u64,
251   ) -> Result<()> {
252 
253     let user = &ctx.accounts.user;
254     let app_data = &ctx.accounts.app_data;
255     let burner = &ctx.accounts.burner;
256 
257     if !burner.is_active {
258       return Err(ErrorCode::Unavailable.into());
259     }
260 
261     let current_timestamp = Clock::get().unwrap().unix_timestamp;
262     let timestamp_per_period = i64::from(app_data.limit) * 3600;
263     let is_in_period = burner.last_period_timestamp + timestamp_per_period <
    current_timestamp;
264     let current_period_burned_amount = if is_in_period {
    burner.per_period_burned_amount } else { 0u64 };
265 
266     if current_period_burned_amount + amount > burner.per_period_burned_limit {
267       return Err(ErrorCode::LimitReached.into());
268     }
269     if burner.total_burned_amount + amount > burner.total_burned_limit {
270       return Err(ErrorCode::LimitReached.into());
271     }
272 
273     let chainlink_program = &ctx.accounts.chainlink_program;
274     let accounts = &ctx.remaining_accounts;
275     let price_feed = &accounts[0];
276     let (price, precision) = get_price_feed(
277         &*chainlink_program,
278         &*price_feed,
279       );
```

```
280     let output_amount = multiply_fraction(amount, precision, price);
281     let output_precision = u64::pow(10, u32::from(burner.output_decimals));
282     let output_amount = multiply_fraction(output_amount, output_precision,
    CUSD_PRECISION);
283
284     let pool_cusd = &ctx.accounts.pool_cusd;
285     let user_cusd = &ctx.accounts.user_cusd;
286     transfer_token(
287         &*user,
288         &user_cusd.to_account_info(),
289         &pool_cusd.to_account_info(),
290         amount,
291         &[],
292       )
293       .expect("CUSD Factory: CPI failed.");
294
295     let root_signer = &ctx.accounts.root_signer;
296     let cusd_mint = &ctx.accounts.cusd_mint;
297     let seeds: &[&[u8]] = &[
298       ROOT_SIGNER_SEED_1,
299       ROOT_SIGNER_SEED_2,
300       &[app_data.signer_nonce],
301     ];
302     burn_token(
303         &*root_signer,
304         &*cusd_mint,
305         &pool_cusd.to_account_info(),
306         amount,
307         &[&seeds],
308       )
309       .expect("CUSD Factory: CPI failed.");
310
311     let burner = &mut ctx.accounts.burner;
312     burner.total_burned_amount = burner.total_burned_amount + amount;
313     burner.per_period_burned_limit = current_period_burned_amount + amount;
314     if !is_in_period {
315       burner.last_period_timestamp = current_timestamp;
316     }
317     let protocol_fee = multiply_fraction(output_amount,
    u64::from(burner.fee_percent), 10000);
318     let amount_to_transfer = output_amount.checked_sub(protocol_fee).unwrap();
319     burner.accumulated_fee =
    burner.accumulated_fee.checked_add(protocol_fee).unwrap();
320
321     let pool_token = &accounts[1];
322     let pool_token =
    TokenAccount::unpack_from_slice(&pool_token.try_borrow_data().unwrap()).unwrap(
```

```
    );
323     if pool_token.owner != root_signer.key() || pool_token.mint !=
    burner.output_token {
324       return Err(ErrorCode::InvalidAccount.into());
325     }
326     let user_token = &accounts[2];
327     let user_token =
    TokenAccount::unpack_from_slice(&user_token.try_borrow_data().unwrap()).unwrap(
    );
328     if user_token.mint != burner.output_token {
329       return Err(ErrorCode::InvalidAccount.into());
330     }
331     transfer_token(
332         &*root_signer,
333         &accounts[1],
334         &accounts[2],
335         amount_to_transfer,
336         &[&seeds],
337       )
338       .expect("CUSD Factory: CPI failed.");
339
340     Ok(())
341 }
```

Please note that the remediation for other issues are not yet applied in the examples above.

5.3. Upgradability of Solana Program

ID	IDX-002
Target	coin98_dollar_mint_burn
Category	General Smart Contract Vulnerability
CWE	CWE-284: Improper Access Control
Risk	**Severity: High** **Impact: High** The logic of the affected programs can be arbitrarily changed. This allows the upgrade authority to change the logic of the program in favor to the platform, e.g., transferring the users' funds to the platform owner's account. **Likelihood: Medium** Only the program upgrade authority can redeploy the program to the same program address; however, there is no restriction to prevent the authority from inserting malicious logic.
Status	Resolved * The Coin98 team has mitigated this issue by confirming that the upgrade authority will be a multisig account controlled by multiple trusted parties.

5.3.1. Description

Programs on Solana can be deployed through the upgradable BPF loader to make them upgradable, allowing the program's upgrade authority to redeploy the program with the new logic, bug fixes, or upgrades to the same program address.

However, there is no restriction on how and when the program will be upgraded. This opens up an attack surface on the program, allowing the upgrade authority to redeploy the program with malicious logic and gain unfair benefits from the users, for example, transferring funds out from the users' accounts.

5.3.2. Remediation

Inspex suggests deploying the program as an immutable program to prevent the program logic from being modified.

However, if the upgradability is needed, Inspex suggests mitigating this issue by the following options:

- Using a multisig account controlled by multiple trusted parties as the upgrade authority
- Implementing a community-run governance to control the redeployment of the program

5.4. Centralized Control of State Variable

ID	IDX-003
Target	coin98_dollar_mint_burn
Category	General Smart Contract Vulnerability
CWE	CWE-284: Improper Access Control
Risk	**Severity: High** **Impact: High** The controlling authorities can change the critical state variables to gain additional profit. Thus, it is unfair to the other users. **Likelihood: Medium** There is nothing to restrict the changes from being done; however, this action can only be done by the contract owner.
Status	Resolved * The Coin98 team has mitigated this issue by confirming that they will use the multisig account as an authorized party to ensure that all privilege contracts are well prepared since the multisig account's execution requires that a list of members in the authorized party must agree.

5.4.1. 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:

File	Program	Access Control	Function
/programs/coin98_dollar_mint_burn/src/lib.rs (L:59)	coin98_dollar_mint_burn	is_root	set_minter()
/programs/coin98_dollar_mint_burn/src/lib.rs (L:115)	coin98_dollar_mint_burn	is_root	set_burner()
/programs/coin98_dollar_mint_burn/src/lib.rs (L:342)	coin98_dollar_mint_burn	is_root	withdraw_token()
/programs/coin98_dollar_mint_b	coin98_dollar_mint_burn	is_root	unlock_token_mint()

urn/src/lib.rs (L:370)			
/programs/coin98_dollar_mint_burn/src/lib.rs (L:416)	coin98_dollar_mint_burn	is_root	set_app_data()

5.4.2. Remediation

In the ideal case, the critical state variables should not be modifiable to keep the integrity of the program. However, if modifications are needed, Inspex suggests limiting the use of these functions by the following options:

- Using a multisig account controlled by multiple trusted parties to ensure that the changes of critical states are well prepared
- Implementing a community-run governance to control the use of these functions

5.5. Design Flaw in cUSD Token

ID	IDX-004
Target	coin98_dollar_mint_burn
Category	Advanced Smart Contract Vulnerability
CWE	CWE-701: Weaknesses Introduced During Design
Risk	**Severity: High** **Impact: High** When the reserve in the `pool_cusd` account is insufficient, the users cannot burn the cUSD token to receive the asset token back. **Likelihood: Medium** 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 asset back tokens.
Status	**Acknowledged** The Coin98 team has acknowledged and accepted the issue's risk and has confirmed that it has prepared a pool to cover the loss in case the contract's funds are insufficient.

5.5.1. Description

The user can execute the `mint()` function to transfer the asset tokens from the user to the account, then the cUSD token will be minted to the user for the same amount as the asset value in USD.

lib.rs

```
142 pub fn mint<'a>(
143     ctx: Context<'_, '_, '_, 'a, MintContext<'a>>,
144     amount: u64,
145     extra_instructions: Vec<u8>,
146   ) -> Result<()> {
147
148     let user = &ctx.accounts.user;
149     let app_data = &ctx.accounts.app_data;
150     let root_signer = &ctx.accounts.root_signer;
151     let minter = &ctx.accounts.minter;
152
153     if !minter.is_active {
154       return Err(ErrorCode::Unavailable.into());
155     }
156
157     let current_timestamp = Clock::get().unwrap().unix_timestamp;
158     let timestamp_per_period = i64::from(app_data.limit) * 3600;
159     let is_in_period = minter.last_period_timestamp + timestamp_per_period <
```

```
current_timestamp;
160     let current_period_minted_amount = if is_in_period { minter.per_period_minted_amount } else { 0u64 };
161
162     if current_period_minted_amount + amount > minter.per_period_minted_limit {
163       return Err(ErrorCode::LimitReached.into());
164     }
165     if minter.total_minted_amount + amount > minter.total_minted_limit {
166       return Err(ErrorCode::LimitReached.into());
167     }
168
169     let chainlink_program = &ctx.accounts.chainlink_program;
170     let accounts = &ctx.remaining_accounts;
171
172     let account_indices: Vec<usize> = extra_instructions.iter()
173       .map(|extra| {
174         usize::from(*extra)
175       })
176       .collect();
177
178     for (i, input_token) in minter.input_tokens.iter().enumerate() {
179       let input_price_feed = &minter.input_price_feeds[i];
180       let price_feed = &accounts[3*i];
181       if price_feed.key() != *input_price_feed {
182         return Err(ErrorCode::InvalidAccount.into());
183       }
184       let (price, precision) = get_price_feed(
185           &*chainlink_program,
186           &*price_feed,
187         );
188       let value_contrib = minter.input_percentages[i];
189
190       let input_vaule = amount.checked_mul(u64::from(value_contrib)).unwrap().checked_div(10000).unwrap();
191       let input_amount = multiply_fraction(input_vaule, precision, price);
192
193       let from_account_index = account_indices[3*i + 1];
194       let to_account_index = account_indices[3*i + 2];
195       let from_account = &accounts[from_account_index];
196       let from_account = TokenAccount::unpack_from_slice(&from_account.try_borrow_data().unwrap()).unwrap();
197       let to_account = &accounts[to_account_index];
198       let to_account = TokenAccount::unpack_from_slice(&to_account.try_borrow_data().unwrap()).unwrap();
```

```
199       if from_account.mint != *input_token {
200         return Err(ErrorCode::InvalidAccount.into());
201       }
202       if to_account.mint != *input_token || to_account.owner !=
root_signer.key() {
203         return Err(ErrorCode::InvalidAccount.into());
204       }
205
206       transfer_token(
207           &*user,
208           &accounts[from_account_index],
209           &accounts[to_account_index],
210           input_amount,
211           &[],
212         )
213         .expect("CUSD Factory: CPI failed.");
214     }
215
216     let minter = &mut ctx.accounts.minter;
217     minter.total_minted_amount = minter.total_minted_amount + amount;
218     minter.per_period_minted_limit = current_period_minted_amount + amount;
219     if !is_in_period {
220       minter.last_period_timestamp = current_timestamp;
221     }
222
223     let protocol_fee = multiply_fraction(amount, u64::from(minter.fee_percent),
10000);
224     let amount_to_transfer = amount.checked_sub(protocol_fee).unwrap();
225     minter.accumulated_fee =
minter.accumulated_fee.checked_add(protocol_fee).unwrap();
226
227     let cusd_mint = &ctx.accounts.cusd_mint;
228     let recipient = &ctx.accounts.recipient;
229
230     let seeds: &[&[u8]] = &[
231       ROOT_SIGNER_SEED_1,
232       ROOT_SIGNER_SEED_2,
233       &[app_data.signer_nonce],
234     ];
235
236     mint_token(
237         &*root_signer,
238         &*cusd_mint,
239         &*recipient,
240         amount_to_transfer,
241         &[&seeds],
242       )
```

```
243         .expect("CUSD Factory: CPI failed.");
244
245     Ok(())
246   }
```

The user can execute the **burn()** function to burn the cUSD token, then the user will receive only one type of asset token with the same USD value.

lib.rs

```
248 pub fn burn<'a>(
249     ctx: Context<'_, '_, '_, 'a, BurnContext<'a>>,
250     amount: u64,
251   ) -> Result<()> {
252
253     let user = &ctx.accounts.user;
254     let app_data = &ctx.accounts.app_data;
255     let burner = &ctx.accounts.burner;
256
257     if !burner.is_active {
258       return Err(ErrorCode::Unavailable.into());
259     }
260
261     let current_timestamp = Clock::get().unwrap().unix_timestamp;
262     let timestamp_per_period = i64::from(app_data.limit) * 3600;
263     let is_in_period = burner.last_period_timestamp + timestamp_per_period <
    current_timestamp;
264     let current_period_burned_amount = if is_in_period {
    burner.per_period_burned_amount } else { 0u64 };
265
266     if current_period_burned_amount + amount > burner.per_period_burned_limit {
267       return Err(ErrorCode::LimitReached.into());
268     }
269     if burner.total_burned_amount + amount > burner.total_burned_limit {
270       return Err(ErrorCode::LimitReached.into());
271     }
272
273     let chainlink_program = &ctx.accounts.chainlink_program;
274     let accounts = &ctx.remaining_accounts;
275     let price_feed = &accounts[0];
276     let (price, precision) = get_price_feed(
277         &*chainlink_program,
278         &*price_feed,
279       );
280     let output_amount = multiply_fraction(amount, precision, price);
281
282     let pool_cusd = &ctx.accounts.pool_cusd;
283     let user_cusd = &ctx.accounts.user_cusd;
```

```
284     transfer_token(
285         &*user,
286         &user_cusd.to_account_info(),
287         &pool_cusd.to_account_info(),
288         amount,
289         &[],
290       )
291       .expect("CUSD Factory: CPI failed.");
292
293     let root_signer = &ctx.accounts.root_signer;
294     let cusd_mint = &ctx.accounts.cusd_mint;
295     let seeds: &[&[u8]] = &[
296       ROOT_SIGNER_SEED_1,
297       ROOT_SIGNER_SEED_2,
298       &[app_data.signer_nonce],
299     ];
300     burn_token(
301         &*root_signer,
302         &*cusd_mint,
303         &pool_cusd.to_account_info(),
304         amount,
305         &[&seeds],
306       )
307       .expect("CUSD Factory: CPI failed.");
308
309     let burner = &mut ctx.accounts.burner;
310     burner.total_burned_amount = burner.total_burned_amount + amount;
311     burner.per_period_burned_limit = current_period_burned_amount + amount;
312     if !is_in_period {
313       burner.last_period_timestamp = current_timestamp;
314     }
315     let protocol_fee = multiply_fraction(output_amount,
    u64::from(burner.fee_percent), 10000);
316     let amount_to_transfer = output_amount.checked_sub(protocol_fee).unwrap();
317     burner.accumulated_fee =
    burner.accumulated_fee.checked_add(protocol_fee).unwrap();
318
319     let pool_token = &accounts[1];
320     let pool_token =
    TokenAccount::unpack_from_slice(&pool_token.try_borrow_data().unwrap()).unwrap(
    );
321     if pool_token.owner != root_signer.key() || pool_token.mint !=
    burner.output_token {
322       return Err(ErrorCode::InvalidAccount.into());
323     }
324     let user_token = &accounts[2];
325     let user_token =
```

```
    TokenAccount::unpack_from_slice(&user_token.try_borrow_data().unwrap()).unwrap(
    );
326     if user_token.mint != burner.output_token {
327       return Err(ErrorCode::InvalidAccount.into());
328     }
329     transfer_token(
330         &*root_signer,
331         &accounts[1],
332         &accounts[2],
333         amount_to_transfer,
334         &[&seeds],
335       )
336       .expect("CUSD Factory: CPI failed.");
337
338     Ok(())
339   }
```

In case the reserve collateral is insufficient, the user cannot burn the cUSD token to get the asset tokens back at lines 329-336, for example:

The user transfers the C98 token in 10$ and the USDC token in 90$ to mint the 100 cUSD token; however, the user can burn the cUSD token to get only one type of the asset token back and the remaining asset token will be insufficient in the account. In case the owner does not transfer sufficient reserve to the account. This results in the users being unable to burn the cUSD token to get the asset tokens back.

5.5.2. Remediation

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

- Fully asset-backed with stable coin.

  The cUSD token 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 token should be backed by the collateral asset which is worth more than the cUSD token minted amount in USD. The Over collateral mechanism must be implemented along with the liquidation mechanism for preventing the bad debt for the platform.

5.6. Incorrect Logic Operator

ID	IDX-005
Target	coin98_dollar_mint_burn
Category	Advanced Smart Contract Vulnerability
CWE	CWE-840: Business Logic Errors
Risk	**Severity:** Medium **Impact:** Low The `per_period_minted_limit` variable in both `minter` and `burner` can not be updated. It causes the per-period mint and burn amount limitation mechanism to be ineffective. **Likelihood:** High The implemented logic for setting the `is_in_period` variable will most likely result in `true`, which will cause the `per_period_minted_limit` variable to not be updated.
Status	Resolved The Coin98 team has resolved this issue by correcting the logic operator in the `mint()` and `burn()` functions. This issue has been resolved in commit `dbea58d618c700be8f9734885a8d9433c7e61e09`.

5.6.1. Description

In the minting and burning procedures in the `coin98_dollar_mint_burn` program, they have implemented a mechanism to limit the amount of the cUSD tokens that are minted and burned in each period. The length of each period is defined by the `timestamp_per_period` variable.

The `is_in_period` variable at line 159 is a check whether the transaction has entered a new period yet. Despite the name, `is_in_period`, the current implemented logic is checking oppositely. It will be `true` if the current transaction's timestamp has passed the latest period; otherwise it will be `false`. So, the `is_in_period` variable will be `true`, if it is a new period and will be `false` if it is in the current period.

The logic to determine the period in the minting procedure is the following expression, `minter.last_period_timestamp + timestamp_per_period`, the `last_period_timestamp` variable will be updated at line 220 with a condition at line 219. The `last_period_timestamp` variable will be updated only when the `is_in_period` is `false`. For now, the only case where the `is_in_period` variable is `false` is that the value of the `current_timestamp` variable has to be lower or equal to the result of `minter.last_period_timestamp + timestamp_per_period`.

Since the `minter.last_period_timestamp` has never been set before, the value will be `0` for the first time. So, to update the `minter.last_period_timestamp` for the first time, the `timestamp_per_period` value must be more than or equal to the `current_timestamp` value, which is a large amount of the period length.

If the transaction fails to update the `last_period_timestamp` for the first time, the following transactions will surely continue to fail to update. The issue also refers to the burning procedure as well.

lib.rs

```
142 pub fn mint<'a>(
143 ctx: Context<'_, '_, '_, 'a, MintContext<'a>>,
144 amount: u64,
145 extra_instructions: Vec<u8>,
146 ) -> Result<()> {
147
148 let user = &ctx.accounts.user;
149 let app_data = &ctx.accounts.app_data;
150 let root_signer = &ctx.accounts.root_signer;
151 let minter = &ctx.accounts.minter;
152
153 if !minter.is_active {
154   return Err(ErrorCode::Unavailable.into());
155 }
156
157 let current_timestamp = Clock::get().unwrap().unix_timestamp;
158 let timestamp_per_period = i64::from(app_data.limit) * 3600;
159 let is_in_period = minter.last_period_timestamp + timestamp_per_period <
    current_timestamp;
160 let current_period_minted_amount = if is_in_period {
    minter.per_period_minted_amount } else { 0u64 };
161
162 if current_period_minted_amount + amount > minter.per_period_minted_limit {
163   return Err(ErrorCode::LimitReached.into());
164 }
165 if minter.total_minted_amount + amount > minter.total_minted_limit {
166   return Err(ErrorCode::LimitReached.into());
167 }
168
169 let chainlink_program = &ctx.accounts.chainlink_program;
170 let accounts = &ctx.remaining_accounts;
171
172 let account_indices: Vec<usize> = extra_instructions.iter()
173   .map(|extra| {
174     usize::from(*extra)
175   })
176   .collect();
177
178 for (i, input_token) in minter.input_tokens.iter().enumerate() {
179   let input_price_feed = &minter.input_price_feeds[i];
180   let price_feed = &accounts[3*i];
181   if price_feed.key() != *input_price_feed {
182     return Err(ErrorCode::InvalidAccount.into());
```

```
  }
  let (price, precision) = get_price_feed(
      &*chainlink_program,
      &*price_feed,
    );
  let value_contrib = minter.input_percentages[i];

  let input_vaule =
amount.checked_mul(u64::from(value_contrib)).unwrap().checked_div(10000).unwrap
();
  let input_amount = multiply_fraction(input_vaule, precision, price);

  let from_account_index = account_indices[3*i + 1];
  let to_account_index = account_indices[3*i + 2];
  let from_account = &accounts[from_account_index];
  let from_account =
TokenAccount::unpack_from_slice(&from_account.try_borrow_data().unwrap()).unwra
p();
  let to_account = &accounts[to_account_index];
  let to_account =
TokenAccount::unpack_from_slice(&to_account.try_borrow_data().unwrap()).unwrap(
);
  if from_account.mint != *input_token {
    return Err(ErrorCode::InvalidAccount.into());
  }
  if to_account.mint != *input_token || to_account.owner != root_signer.key() {
    return Err(ErrorCode::InvalidAccount.into());
  }

  transfer_token(
      &*user,
      &accounts[from_account_index],
      &accounts[to_account_index],
      input_amount,
      &[],
    )
    .expect("CUSD Factory: CPI failed.");
}

let minter = &mut ctx.accounts.minter;
minter.total_minted_amount = minter.total_minted_amount + amount;
minter.per_period_minted_limit = current_period_minted_amount + amount;
if !is_in_period {
  minter.last_period_timestamp = current_timestamp;
}

let protocol_fee = multiply_fraction(amount, u64::from(minter.fee_percent),
```

```
    10000);
224 let amount_to_transfer = amount.checked_sub(protocol_fee).unwrap();
225 minter.accumulated_fee =
    minter.accumulated_fee.checked_add(protocol_fee).unwrap();
226
227 let cusd_mint = &ctx.accounts.cusd_mint;
228 let recipient = &ctx.accounts.recipient;
229
230 let seeds: &[&[u8]] = &[
231   ROOT_SIGNER_SEED_1,
232   ROOT_SIGNER_SEED_2,
233   &[app_data.signer_nonce],
234 ];
235
236 mint_token(
237     &*root_signer,
238     &*cusd_mint,
239     &*recipient,
240     amount_to_transfer,
241     &[&seeds],
242   )
243   .expect("CUSD Factory: CPI failed.");
244
245 Ok(())
246 }
```

5.6.2. Remediation

Inspex suggests changing the logical operation in line 159 for the `mint()` function from lesser than (<) to greater than (>) or `greater than or equal to` (>=) to check the condition as intended.

lib.rs

```
142 pub fn mint<'a>(
143 ctx: Context<'_, '_, '_, 'a, MintContext<'a>>,
144 amount: u64,
145 extra_instructions: Vec<u8>,
146 ) -> Result<()> {
147
148 let user = &ctx.accounts.user;
149 let app_data = &ctx.accounts.app_data;
150 let root_signer = &ctx.accounts.root_signer;
151 let minter = &ctx.accounts.minter;
152
153 if !minter.is_active {
154   return Err(ErrorCode::Unavailable.into());
155 }
156
```

```
157 let current_timestamp = Clock::get().unwrap().unix_timestamp;
158 let timestamp_per_period = i64::from(app_data.limit) * 3600;
159 let is_in_period = minter.last_period_timestamp + timestamp_per_period >
    current_timestamp;
160 let current_period_minted_amount = if is_in_period {
    minter.per_period_minted_amount } else { 0u64 };
161
162 if current_period_minted_amount + amount > minter.per_period_minted_limit {
163   return Err(ErrorCode::LimitReached.into());
164 }
165 if minter.total_minted_amount + amount > minter.total_minted_limit {
166   return Err(ErrorCode::LimitReached.into());
167 }
168
169 let chainlink_program = &ctx.accounts.chainlink_program;
170 let accounts = &ctx.remaining_accounts;
171
172 let account_indices: Vec<usize> = extra_instructions.iter()
173   .map(|extra| {
174     usize::from(*extra)
175   })
176   .collect();
177
178 for (i, input_token) in minter.input_tokens.iter().enumerate() {
179   let input_price_feed = &minter.input_price_feeds[i];
180   let price_feed = &accounts[3*i];
181   if price_feed.key() != *input_price_feed {
182     return Err(ErrorCode::InvalidAccount.into());
183   }
184   let (price, precision) = get_price_feed(
185       &*chainlink_program,
186       &*price_feed,
187     );
188   let value_contrib = minter.input_percentages[i];
189
190   let input_vaule =
    amount.checked_mul(u64::from(value_contrib)).unwrap().checked_div(10000).unwrap
    ();
191   let input_amount = multiply_fraction(input_vaule, precision, price);
192
193   let from_account_index = account_indices[3*i + 1];
194   let to_account_index = account_indices[3*i + 2];
195   let from_account = &accounts[from_account_index];
196   let from_account =
    TokenAccount::unpack_from_slice(&from_account.try_borrow_data().unwrap()).unwra
    p();
197   let to_account = &accounts[to_account_index];
```

```
198   let to_account =
      TokenAccount::unpack_from_slice(&to_account.try_borrow_data().unwrap()).unwrap(
      );
199   if from_account.mint != *input_token {
200     return Err(ErrorCode::InvalidAccount.into());
201   }
202   if to_account.mint != *input_token || to_account.owner != root_signer.key() {
203     return Err(ErrorCode::InvalidAccount.into());
204   }
205
206   transfer_token(
207       &*user,
208       &accounts[from_account_index],
209       &accounts[to_account_index],
210       input_amount,
211       &[],
212     )
213     .expect("CUSD Factory: CPI failed.");
214 }
215
216 let minter = &mut ctx.accounts.minter;
217 minter.total_minted_amount = minter.total_minted_amount + amount;
218 minter.per_period_minted_limit = current_period_minted_amount + amount;
219 if !is_in_period {
220   minter.last_period_timestamp = current_timestamp;
221 }
222
223 let protocol_fee = multiply_fraction(amount, u64::from(minter.fee_percent),
      10000);
224 let amount_to_transfer = amount.checked_sub(protocol_fee).unwrap();
225 minter.accumulated_fee =
      minter.accumulated_fee.checked_add(protocol_fee).unwrap();
226
227 let cusd_mint = &ctx.accounts.cusd_mint;
228 let recipient = &ctx.accounts.recipient;
229
230 let seeds: &[&[u8]] = &[
231   ROOT_SIGNER_SEED_1,
232   ROOT_SIGNER_SEED_2,
233   &[app_data.signer_nonce],
234 ];
235
236 mint_token(
237     &*root_signer,
238     &*cusd_mint,
239     &*recipient,
240     amount_to_transfer,
```

```
241       &[&seeds],
242     )
243     .expect("CUSD Factory: CPI failed.");
244
245 Ok(())
246 }
```

For the `burn()` function, the logical operator in line 263 should be changed as well.

lib.rs

```
248 pub fn burn<'a>(
249 ctx: Context<'_, '_, '_, 'a, BurnContext<'a>>,
250 amount: u64,
251 ) -> Result<()> {
252
253 let user = &ctx.accounts.user;
254 let app_data = &ctx.accounts.app_data;
255 let burner = &ctx.accounts.burner;
256
257 if !burner.is_active {
258   return Err(ErrorCode::Unavailable.into());
259 }
260
261 let current_timestamp = Clock::get().unwrap().unix_timestamp;
262 let timestamp_per_period = i64::from(app_data.limit) * 3600;
263 let is_in_period = burner.last_period_timestamp + timestamp_per_period >
    current_timestamp;
264 let current_period_burned_amount = if is_in_period {
    burner.per_period_burned_amount } else { 0u64 };
265
266 if current_period_burned_amount + amount > burner.per_period_burned_limit {
267   return Err(ErrorCode::LimitReached.into());
268 }
269 if burner.total_burned_amount + amount > burner.total_burned_limit {
270   return Err(ErrorCode::LimitReached.into());
271 }
272
273 let chainlink_program = &ctx.accounts.chainlink_program;
274 let accounts = &ctx.remaining_accounts;
275 let price_feed = &accounts[0];
276 let (price, precision) = get_price_feed(
277     &*chainlink_program,
278     &*price_feed,
279   );
280 let output_amount = multiply_fraction(amount, precision, price);
281
282 let pool_cusd = &ctx.accounts.pool_cusd;
```

```
283 let user_cusd = &ctx.accounts.user_cusd;
284 transfer_token(
285     &*user,
286     &user_cusd.to_account_info(),
287     &pool_cusd.to_account_info(),
288     amount,
289     &[],
290   )
291   .expect("CUSD Factory: CPI failed.");
292
293 let root_signer = &ctx.accounts.root_signer;
294 let cusd_mint = &ctx.accounts.cusd_mint;
295 let seeds: &[&[u8]] = &[
296   ROOT_SIGNER_SEED_1,
297   ROOT_SIGNER_SEED_2,
298   &[app_data.signer_nonce],
299 ];
300 burn_token(
301     &*root_signer,
302     &*cusd_mint,
303     &pool_cusd.to_account_info(),
304     amount,
305     &[&seeds],
306   )
307   .expect("CUSD Factory: CPI failed.");
308
309 let burner = &mut ctx.accounts.burner;
310 burner.total_burned_amount = burner.total_burned_amount + amount;
311 burner.per_period_burned_limit = current_period_burned_amount + amount;
312 if !is_in_period {
313   burner.last_period_timestamp = current_timestamp;
314 }
315 let protocol_fee = multiply_fraction(output_amount,
    u64::from(burner.fee_percent), 10000);
316 let amount_to_transfer = output_amount.checked_sub(protocol_fee).unwrap();
317 burner.accumulated_fee =
    burner.accumulated_fee.checked_add(protocol_fee).unwrap();
318
319 let pool_token = &accounts[1];
320 let pool_token =
    TokenAccount::unpack_from_slice(&pool_token.try_borrow_data().unwrap()).unwrap(
    );
321 if pool_token.owner != root_signer.key() || pool_token.mint !=
    burner.output_token {
322   return Err(ErrorCode::InvalidAccount.into());
323 }
324 let user_token = &accounts[2];
```

```
let user_token =
TokenAccount::unpack_from_slice(&user_token.try_borrow_data().unwrap()).unwrap(
);
if user_token.mint != burner.output_token {
  return Err(ErrorCode::InvalidAccount.into());
}
transfer_token(
    &*root_signer,
    &accounts[1],
    &accounts[2],
    amount_to_transfer,
    &[&seeds],
  )
  .expect("CUSD Factory: CPI failed.");

Ok(())
}
```

Please note that the remediation for other issues are not yet applied in the examples above.

5.7. Incorrect Update Account State

ID	IDX-006
Target	coin98_dollar_mint_burn
Category	Advanced Smart Contract Vulnerability
CWE	CWE-840: Business Logic Errors
Risk	**Severity:** Medium **Impact:** Low The period limitation mechanism will become ineffective because the mint and burn per period limitation will increase each time mint and burn functions are executed. **Likelihood:** High It is very likely that this issue will occur because the mint and burn functions are the primary function of the platform; every platform user will execute this function.
Status	Resolved The Coin98 team has resolved this issue by updating the `minter.per_period_minted_amount` and `burner.per_period_burned_amount` states every time the `mint()` and `burn()` functions are executed. This issue has been resolved in commit `8ecaa951a2e0b86072631055f1cb4beafc7d716b`.

5.7.1. Description

In the minting and burning procedures in the `coin98_dollar_mint_burn` program, they have implemented a mechanism to limit the amount of the cUSD token that is being minted and burned in each period of time. The total value of the minted and burned amount in each period of time should be stored in the `minter.per_period_minted_amount` and `burner.per_period_burned_amount` respectively.

First, the `coin98_dollar_mint_burn` program checks whether the minted and burned amount in the current period does not exceed the limit amount, for example, the `mint()` function at the lines 160-164.

While the `mint()` or `burn()` function is executing, the value of the `per_period_minted_amount` or the `per_period_minted_amount` should be increased with the input amount. However, the program updates the `per_period_minted_amount` or the `per_period_minted_amount` instead, causing the period limitation mechanism to be inefficient as shown in line 218.

lib.rs

```
142 pub fn mint<'a>(
143   ctx: Context<'_, '_, '_, 'a, MintContext<'a>>,
144   amount: u64,
145   extra_instructions: Vec<u8>,
```

```
146 ) -> Result<()> {
147
148   let user = &ctx.accounts.user;
149   let app_data = &ctx.accounts.app_data;
150   let root_signer = &ctx.accounts.root_signer;
151   let minter = &ctx.accounts.minter;
152
153   if !minter.is_active {
154     return Err(ErrorCode::Unavailable.into());
155   }
156
157   let current_timestamp = Clock::get().unwrap().unix_timestamp;
158   let timestamp_per_period = i64::from(app_data.limit) * 3600;
159   let is_in_period = minter.last_period_timestamp + timestamp_per_period <
    current_timestamp;
160   let current_period_minted_amount = if is_in_period {
    minter.per_period_minted_amount } else { 0u64 };
161
162   if current_period_minted_amount + amount > minter.per_period_minted_limit {
163     return Err(ErrorCode::LimitReached.into());
164   }
165   if minter.total_minted_amount + amount > minter.total_minted_limit {
166     return Err(ErrorCode::LimitReached.into());
167   }
168
169   let chainlink_program = &ctx.accounts.chainlink_program;
170   let accounts = &ctx.remaining_accounts;
171
172   let account_indices: Vec<usize> = extra_instructions.iter()
173     .map(|extra| {
174       usize::from(*extra)
175     })
176     .collect();
177
178   for (i, input_token) in minter.input_tokens.iter().enumerate() {
179     let input_price_feed = &minter.input_price_feeds[i];
180     let price_feed = &accounts[3*i];
181     if price_feed.key() != *input_price_feed {
182       return Err(ErrorCode::InvalidAccount.into());
183     }
184     let (price, precision) = get_price_feed(
185         &*chainlink_program,
186         &*price_feed,
187       );
188     let value_contrib = minter.input_percentages[i];
189
190     let input_vaule =
```

```
amount.checked_mul(u64::from(value_contrib)).unwrap().checked_div(10000).unwrap
();
191     let input_amount = multiply_fraction(input_vaule, precision, price);
192
193     let from_account_index = account_indices[3*i + 1];
194     let to_account_index = account_indices[3*i + 2];
195     let from_account = &accounts[from_account_index];
196     let from_account =
TokenAccount::unpack_from_slice(&from_account.try_borrow_data().unwrap()).unwra
p();
197     let to_account = &accounts[to_account_index];
198     let to_account =
TokenAccount::unpack_from_slice(&to_account.try_borrow_data().unwrap()).unwrap(
);
199     if from_account.mint != *input_token {
200       return Err(ErrorCode::InvalidAccount.into());
201     }
202     if to_account.mint != *input_token || to_account.owner != root_signer.key()
{
203       return Err(ErrorCode::InvalidAccount.into());
204     }
205
206     transfer_token(
207         &*user,
208         &accounts[from_account_index],
209         &accounts[to_account_index],
210         input_amount,
211         &[],
212       )
213       .expect("CUSD Factory: CPI failed.");
214   }
215
216   let minter = &mut ctx.accounts.minter;
217   minter.total_minted_amount = minter.total_minted_amount + amount;
218   minter.per_period_minted_limit = current_period_minted_amount + amount;
219   if !is_in_period {
220     minter.last_period_timestamp = current_timestamp;
221   }
222
223   let protocol_fee = multiply_fraction(amount, u64::from(minter.fee_percent),
10000);
224   let amount_to_transfer = amount.checked_sub(protocol_fee).unwrap();
225   minter.accumulated_fee =
minter.accumulated_fee.checked_add(protocol_fee).unwrap();
226
227   let cusd_mint = &ctx.accounts.cusd_mint;
228   let recipient = &ctx.accounts.recipient;
```

```
229
230   let seeds: &[&[u8]] = &[
231     ROOT_SIGNER_SEED_1,
232     ROOT_SIGNER_SEED_2,
233     &[app_data.signer_nonce],
234   ];
235
236   mint_token(
237       &*root_signer,
238       &*cusd_mint,
239       &*recipient,
240       amount_to_transfer,
241       &[&seeds],
242     )
243     .expect("CUSD Factory: CPI failed.");
244
245   Ok(())
246 }
```

The `per_period_burned_limit` should also be updated in line 311 in the `burn()` function.

lib.rs

```
248 pub fn burn<'a>(
249   ctx: Context<'_, '_, '_, 'a, BurnContext<'a>>,
250   amount: u64,
251 ) -> Result<()> {
252
253   let user = &ctx.accounts.user;
254   let app_data = &ctx.accounts.app_data;
255   let burner = &ctx.accounts.burner;
256
257   if !burner.is_active {
258     return Err(ErrorCode::Unavailable.into());
259   }
260
261   let current_timestamp = Clock::get().unwrap().unix_timestamp;
262   let timestamp_per_period = i64::from(app_data.limit) * 3600;
263   let is_in_period = burner.last_period_timestamp + timestamp_per_period <
    current_timestamp;
264   let current_period_burned_amount = if is_in_period {
    burner.per_period_burned_amount } else { 0u64 };
265
266   if current_period_burned_amount + amount > burner.per_period_burned_limit {
267     return Err(ErrorCode::LimitReached.into());
268   }
269   if burner.total_burned_amount + amount > burner.total_burned_limit {
270     return Err(ErrorCode::LimitReached.into());
```

```
271   }
272
273   let chainlink_program = &ctx.accounts.chainlink_program;
274   let accounts = &ctx.remaining_accounts;
275   let price_feed = &accounts[0];
276   let (price, precision) = get_price_feed(
277       &*chainlink_program,
278       &*price_feed,
279     );
280   let output_amount = multiply_fraction(amount, precision, price);
281
282   let pool_cusd = &ctx.accounts.pool_cusd;
283   let user_cusd = &ctx.accounts.user_cusd;
284   transfer_token(
285       &*user,
286       &user_cusd.to_account_info(),
287       &pool_cusd.to_account_info(),
288       amount,
289       &[],
290     )
291     .expect("CUSD Factory: CPI failed.");
292
293   let root_signer = &ctx.accounts.root_signer;
294   let cusd_mint = &ctx.accounts.cusd_mint;
295   let seeds: &[&[u8]] = &[
296     ROOT_SIGNER_SEED_1,
297     ROOT_SIGNER_SEED_2,
298     &[app_data.signer_nonce],
299   ];
300   burn_token(
301       &*root_signer,
302       &*cusd_mint,
303       &pool_cusd.to_account_info(),
304       amount,
305       &[&seeds],
306     )
307     .expect("CUSD Factory: CPI failed.");
308
309   let burner = &mut ctx.accounts.burner;
310   burner.total_burned_amount = burner.total_burned_amount + amount;
311   burner.per_period_burned_limit = current_period_burned_amount + amount;
312   if !is_in_period {
313     burner.last_period_timestamp = current_timestamp;
314   }
315   let protocol_fee = multiply_fraction(output_amount,
    u64::from(burner.fee_percent), 10000);
316   let amount_to_transfer = output_amount.checked_sub(protocol_fee).unwrap();
```

```
317   burner.accumulated_fee =
      burner.accumulated_fee.checked_add(protocol_fee).unwrap();
318
319   let pool_token = &accounts[1];
320   let pool_token =
      TokenAccount::unpack_from_slice(&pool_token.try_borrow_data().unwrap()).unwrap(
      );
321   if pool_token.owner != root_signer.key() || pool_token.mint !=
      burner.output_token {
322     return Err(ErrorCode::InvalidAccount.into());
323   }
324   let user_token = &accounts[2];
325   let user_token =
      TokenAccount::unpack_from_slice(&user_token.try_borrow_data().unwrap()).unwrap(
      );
326   if user_token.mint != burner.output_token {
327     return Err(ErrorCode::InvalidAccount.into());
328   }
329   transfer_token(
330       &*root_signer,
331       &accounts[1],
332       &accounts[2],
333       amount_to_transfer,
334       &[&seeds],
335     )
336     .expect("CUSD Factory: CPI failed.");
337
338   Ok(())
339 }
```

5.7.2. Remediation

Inspex suggests updating the value of the `minter.per_period_minted_amount` and `burner.per_period_burned_amount` every minting and burning procedure, for example as shown in lines 218, 311.

lib.rs

```
142 pub fn mint<'a>(
143   ctx: Context<'_, '_, '_, 'a, MintContext<'a>>,
144   amount: u64,
145   extra_instructions: Vec<u8>,
146 ) -> Result<()> {
147
148   let user = &ctx.accounts.user;
149   let app_data = &ctx.accounts.app_data;
150   let root_signer = &ctx.accounts.root_signer;
151   let minter = &ctx.accounts.minter;
```

```
152
153   if !minter.is_active {
154     return Err(ErrorCode::Unavailable.into());
155   }
156
157   let current_timestamp = Clock::get().unwrap().unix_timestamp;
158   let timestamp_per_period = i64::from(app_data.limit) * 3600;
159   let is_in_period = minter.last_period_timestamp + timestamp_per_period <
current_timestamp;
160   let current_period_minted_amount = if is_in_period {
minter.per_period_minted_amount } else { 0u64 };
161
162   if current_period_minted_amount + amount > minter.per_period_minted_limit {
163     return Err(ErrorCode::LimitReached.into());
164   }
165   if minter.total_minted_amount + amount > minter.total_minted_limit {
166     return Err(ErrorCode::LimitReached.into());
167   }
168
169   let chainlink_program = &ctx.accounts.chainlink_program;
170   let accounts = &ctx.remaining_accounts;
171
172   let account_indices: Vec<usize> = extra_instructions.iter()
173     .map(|extra| {
174       usize::from(*extra)
175     })
176     .collect();
177
178   for (i, input_token) in minter.input_tokens.iter().enumerate() {
179     let input_price_feed = &minter.input_price_feeds[i];
180     let price_feed = &accounts[3*i];
181     if price_feed.key() != *input_price_feed {
182       return Err(ErrorCode::InvalidAccount.into());
183     }
184     let (price, precision) = get_price_feed(
185         &*chainlink_program,
186         &*price_feed,
187       );
188     let value_contrib = minter.input_percentages[i];
189
190     let input_vaule =
amount.checked_mul(u64::from(value_contrib)).unwrap().checked_div(10000).unwrap
();
191     let input_amount = multiply_fraction(input_vaule, precision, price);
192
193     let from_account_index = account_indices[3*i + 1];
194     let to_account_index = account_indices[3*i + 2];
```

```
195     let from_account = &accounts[from_account_index];
196     let from_account =
TokenAccount::unpack_from_slice(&from_account.try_borrow_data().unwrap()).unwra
p();
197     let to_account = &accounts[to_account_index];
198     let to_account =
TokenAccount::unpack_from_slice(&to_account.try_borrow_data().unwrap()).unwrap(
);
199     if from_account.mint != *input_token {
200       return Err(ErrorCode::InvalidAccount.into());
201     }
202     if to_account.mint != *input_token || to_account.owner != root_signer.key()
{
203       return Err(ErrorCode::InvalidAccount.into());
204     }
205
206     transfer_token(
207         &*user,
208         &accounts[from_account_index],
209         &accounts[to_account_index],
210         input_amount,
211         &[],
212       )
213       .expect("CUSD Factory: CPI failed.");
214   }
215
216   let minter = &mut ctx.accounts.minter;
217   minter.total_minted_amount = minter.total_minted_amount + amount;
218   minter.per_period_minted_amount = current_period_minted_amount + amount;
219   if !is_in_period {
220     minter.last_period_timestamp = current_timestamp;
221   }
222
223   let protocol_fee = multiply_fraction(amount, u64::from(minter.fee_percent),
10000);
224   let amount_to_transfer = amount.checked_sub(protocol_fee).unwrap();
225   minter.accumulated_fee =
minter.accumulated_fee.checked_add(protocol_fee).unwrap();
226
227   let cusd_mint = &ctx.accounts.cusd_mint;
228   let recipient = &ctx.accounts.recipient;
229
230   let seeds: &[&[u8]] = &[
231     ROOT_SIGNER_SEED_1,
232     ROOT_SIGNER_SEED_2,
233     &[app_data.signer_nonce],
234   ];
```

```

  mint_token(
      &*root_signer,
      &*cusd_mint,
      &*recipient,
      amount_to_transfer,
      &[&seeds],
    )
    .expect("CUSD Factory: CPI failed.");

  Ok(())
}
```

lib.rs

```
pub fn burn<'a>(
  ctx: Context<'_, '_, '_, 'a, BurnContext<'a>>,
  amount: u64,
) -> Result<()> {

  let user = &ctx.accounts.user;
  let app_data = &ctx.accounts.app_data;
  let burner = &ctx.accounts.burner;

  if !burner.is_active {
    return Err(ErrorCode::Unavailable.into());
  }

  let current_timestamp = Clock::get().unwrap().unix_timestamp;
  let timestamp_per_period = i64::from(app_data.limit) * 3600;
  let is_in_period = burner.last_period_timestamp + timestamp_per_period <
current_timestamp;
  let current_period_burned_amount = if is_in_period {
burner.per_period_burned_amount } else { 0u64 };

  if current_period_burned_amount + amount > burner.per_period_burned_limit {
    return Err(ErrorCode::LimitReached.into());
  }
  if burner.total_burned_amount + amount > burner.total_burned_limit {
    return Err(ErrorCode::LimitReached.into());
  }

  let chainlink_program = &ctx.accounts.chainlink_program;
  let accounts = &ctx.remaining_accounts;
  let price_feed = &accounts[0];
  let (price, precision) = get_price_feed(
      &*chainlink_program,
      &*price_feed,
```

```
279       );
280     let output_amount = multiply_fraction(amount, precision, price);
281
282     let pool_cusd = &ctx.accounts.pool_cusd;
283     let user_cusd = &ctx.accounts.user_cusd;
284     transfer_token(
285         &*user,
286         &user_cusd.to_account_info(),
287         &pool_cusd.to_account_info(),
288         amount,
289         &[],
290       )
291       .expect("CUSD Factory: CPI failed.");
292
293     let root_signer = &ctx.accounts.root_signer;
294     let cusd_mint = &ctx.accounts.cusd_mint;
295     let seeds: &[&[u8]] = &[
296       ROOT_SIGNER_SEED_1,
297       ROOT_SIGNER_SEED_2,
298       &[app_data.signer_nonce],
299     ];
300     burn_token(
301         &*root_signer,
302         &*cusd_mint,
303         &pool_cusd.to_account_info(),
304         amount,
305         &[&seeds],
306       )
307       .expect("CUSD Factory: CPI failed.");
308
309     let burner = &mut ctx.accounts.burner;
310     burner.total_burned_amount = burner.total_burned_amount + amount;
311     burner.per_period_burned_amount = current_period_burned_amount + amount;
312     if !is_in_period {
313       burner.last_period_timestamp = current_timestamp;
314     }
315     let protocol_fee = multiply_fraction(output_amount,
    u64::from(burner.fee_percent), 10000);
316     let amount_to_transfer = output_amount.checked_sub(protocol_fee).unwrap();
317     burner.accumulated_fee =
    burner.accumulated_fee.checked_add(protocol_fee).unwrap();
318
319     let pool_token = &accounts[1];
320     let pool_token =
    TokenAccount::unpack_from_slice(&pool_token.try_borrow_data().unwrap()).unwrap(
    );
321     if pool_token.owner != root_signer.key() || pool_token.mint !=
```

```
    burner.output_token {
322     return Err(ErrorCode::InvalidAccount.into());
323   }
324   let user_token = &accounts[2];
325   let user_token =
    TokenAccount::unpack_from_slice(&user_token.try_borrow_data().unwrap()).unwrap(
    );
326   if user_token.mint != burner.output_token {
327     return Err(ErrorCode::InvalidAccount.into());
328   }
329   transfer_token(
330       &*root_signer,
331       &accounts[1],
332       &accounts[2],
333       amount_to_transfer,
334       &[&seeds],
335     )
336     .expect("CUSD Factory: CPI failed.");
337
338   Ok(())
339 }
```

Please note that the remediation for other issues are not yet applied in the examples above.

5.8. Unbound Configuration Parameter

ID	IDX-007
Target	coin98_dollar_mint_burn
Category	Advanced Smart Contract Vulnerability
CWE	CWE-840: Business Logic Errors
Risk	**Severity:** Low **Impact:** Medium If the minting and burning fees are set to 100%, users will pay the entire token amount to the platform fee and receive nothing. **Likelihood:** Low The fee attribute of each minter and burner account can only be set by the owner whose addresses are whitelisted in the `ROOT_KEYS` variable.
Status	Resolved The Coin98 team has resolved this issue by implementing the bound configuration for the system fee. This issue has been resolved in commit `108f982b7bea78680c6fd12a59516ed6076b4b89`.

5.8.1. Description

In each minting and burning procedure, the platform will take a portion of the input tokens as the platform fee is determined by the value set in the `minter` and `burner` accounts as shown in lines 223-225 and 315-317.

lib.rs

```
142 pub fn mint<'a>(
143   ctx: Context<'_, '_, '_, 'a, MintContext<'a>>,
144   amount: u64,
145   extra_instructions: Vec<u8>,
146 ) -> Result<()> {
147
148   let user = &ctx.accounts.user;
149   let app_data = &ctx.accounts.app_data;
150   let root_signer = &ctx.accounts.root_signer;
151   let minter = &ctx.accounts.minter;
152
153   if !minter.is_active {
154     return Err(ErrorCode::Unavailable.into());
155   }
156
```

```
157   let current_timestamp = Clock::get().unwrap().unix_timestamp;
158   let timestamp_per_period = i64::from(app_data.limit) * 3600;
159   let is_in_period = minter.last_period_timestamp + timestamp_per_period <
current_timestamp;
160   let current_period_minted_amount = if is_in_period {
minter.per_period_minted_amount } else { 0u64 };
161
162   if current_period_minted_amount + amount > minter.per_period_minted_limit {
163     return Err(ErrorCode::LimitReached.into());
164   }
165   if minter.total_minted_amount + amount > minter.total_minted_limit {
166     return Err(ErrorCode::LimitReached.into());
167   }
168
169   let chainlink_program = &ctx.accounts.chainlink_program;
170   let accounts = &ctx.remaining_accounts;
171
172   let account_indices: Vec<usize> = extra_instructions.iter()
173     .map(|extra| {
174       usize::from(*extra)
175     })
176     .collect();
177
178   for (i, input_token) in minter.input_tokens.iter().enumerate() {
179     let input_price_feed = &minter.input_price_feeds[i];
180     let price_feed = &accounts[3*i];
181     if price_feed.key() != *input_price_feed {
182       return Err(ErrorCode::InvalidAccount.into());
183     }
184     let (price, precision) = get_price_feed(
185         &*chainlink_program,
186         &*price_feed,
187       );
188     let value_contrib = minter.input_percentages[i];
189
190     let input_vaule =
amount.checked_mul(u64::from(value_contrib)).unwrap().checked_div(10000).unwrap
();
191     let input_amount = multiply_fraction(input_vaule, precision, price);
192
193     let from_account_index = account_indices[3*i + 1];
194     let to_account_index = account_indices[3*i + 2];
195     let from_account = &accounts[from_account_index];
196     let from_account =
TokenAccount::unpack_from_slice(&from_account.try_borrow_data().unwrap()).unwra
p();
197     let to_account = &accounts[to_account_index];
```

```
198     let to_account =
TokenAccount::unpack_from_slice(&to_account.try_borrow_data().unwrap()).unwrap(
);
199     if from_account.mint != *input_token {
200       return Err(ErrorCode::InvalidAccount.into());
201     }
202     if to_account.mint != *input_token || to_account.owner != root_signer.key()
{
203       return Err(ErrorCode::InvalidAccount.into());
204     }
205
206     transfer_token(
207         &*user,
208         &accounts[from_account_index],
209         &accounts[to_account_index],
210         input_amount,
211         &[],
212       )
213       .expect("CUSD Factory: CPI failed.");
214   }
215
216   let minter = &mut ctx.accounts.minter;
217   minter.total_minted_amount = minter.total_minted_amount + amount;
218   minter.per_period_minted_limit = current_period_minted_amount + amount;
219   if !is_in_period {
220     minter.last_period_timestamp = current_timestamp;
221   }
222
223   let protocol_fee = multiply_fraction(amount, u64::from(minter.fee_percent),
10000);
224   let amount_to_transfer = amount.checked_sub(protocol_fee).unwrap();
225   minter.accumulated_fee =
minter.accumulated_fee.checked_add(protocol_fee).unwrap();
226
227   let cusd_mint = &ctx.accounts.cusd_mint;
228   let recipient = &ctx.accounts.recipient;
229
230   let seeds: &[&[u8]] = &[
231     ROOT_SIGNER_SEED_1,
232     ROOT_SIGNER_SEED_2,
233     &[app_data.signer_nonce],
234   ];
235
236   mint_token(
237       &*root_signer,
238       &*cusd_mint,
239       &*recipient,
```

```
240         amount_to_transfer,
241         &[&seeds],
242       )
243       .expect("CUSD Factory: CPI failed.");
244
245     Ok(())
246 }
247
248 pub fn burn<'a>(
249     ctx: Context<'_, '_, '_, 'a, BurnContext<'a>>,
250     amount: u64,
251 ) -> Result<()> {
252
253     let user = &ctx.accounts.user;
254     let app_data = &ctx.accounts.app_data;
255     let burner = &ctx.accounts.burner;
256
257     if !burner.is_active {
258       return Err(ErrorCode::Unavailable.into());
259     }
260
261     let current_timestamp = Clock::get().unwrap().unix_timestamp;
262     let timestamp_per_period = i64::from(app_data.limit) * 3600;
263     let is_in_period = burner.last_period_timestamp + timestamp_per_period <
    current_timestamp;
264     let current_period_burned_amount = if is_in_period {
    burner.per_period_burned_amount } else { 0u64 };
265
266     if current_period_burned_amount + amount > burner.per_period_burned_limit {
267       return Err(ErrorCode::LimitReached.into());
268     }
269     if burner.total_burned_amount + amount > burner.total_burned_limit {
270       return Err(ErrorCode::LimitReached.into());
271     }
272
273     let chainlink_program = &ctx.accounts.chainlink_program;
274     let accounts = &ctx.remaining_accounts;
275     let price_feed = &accounts[0];
276     let (price, precision) = get_price_feed(
277         &*chainlink_program,
278         &*price_feed,
279       );
280     let output_amount = multiply_fraction(amount, precision, price);
281
282     let pool_cusd = &ctx.accounts.pool_cusd;
283     let user_cusd = &ctx.accounts.user_cusd;
284     transfer_token(
```

```
285         &*user,
286         &user_cusd.to_account_info(),
287         &pool_cusd.to_account_info(),
288         amount,
289         &[],
290       )
291       .expect("CUSD Factory: CPI failed.");
292
293     let root_signer = &ctx.accounts.root_signer;
294     let cusd_mint = &ctx.accounts.cusd_mint;
295     let seeds: &[&[u8]] = &[
296       ROOT_SIGNER_SEED_1,
297       ROOT_SIGNER_SEED_2,
298       &[app_data.signer_nonce],
299     ];
300     burn_token(
301         &*root_signer,
302         &*cusd_mint,
303         &pool_cusd.to_account_info(),
304         amount,
305         &[&seeds],
306       )
307       .expect("CUSD Factory: CPI failed.");
308
309     let burner = &mut ctx.accounts.burner;
310     burner.total_burned_amount = burner.total_burned_amount + amount;
311     burner.per_period_burned_limit = current_period_burned_amount + amount;
312     if !is_in_period {
313       burner.last_period_timestamp = current_timestamp;
314     }
315     let protocol_fee = multiply_fraction(output_amount,
    u64::from(burner.fee_percent), 10000);
316     let amount_to_transfer = output_amount.checked_sub(protocol_fee).unwrap();
317     burner.accumulated_fee =
    burner.accumulated_fee.checked_add(protocol_fee).unwrap();
318
319     let pool_token = &accounts[1];
320     let pool_token =
    TokenAccount::unpack_from_slice(&pool_token.try_borrow_data().unwrap()).unwrap(
    );
321     if pool_token.owner != root_signer.key() || pool_token.mint !=
    burner.output_token {
322       return Err(ErrorCode::InvalidAccount.into());
323     }
324     let user_token = &accounts[2];
325     let user_token =
    TokenAccount::unpack_from_slice(&user_token.try_borrow_data().unwrap()).unwrap(
```

```
    );
326   if user_token.mint != burner.output_token {
327     return Err(ErrorCode::InvalidAccount.into());
328   }
329   transfer_token(
330       &*root_signer,
331       &accounts[1],
332       &accounts[2],
333       amount_to_transfer,
334       &[&seeds],
335     )
336     .expect("CUSD Factory: CPI failed.");
337
338   Ok(())
339 }
```

However, the platform owner is able to configure the `minter.fee_percent` and `burner.fee_percent` up to 100%, which causes the user's entire token amount to be paid as the platform fee, then users will receive nothing.

lib.rs

```
58 #[access_control(is_root(*ctx.accounts.root.key))]
59 pub fn set_minter(
60   ctx: Context<SetMinterContext>,
61   is_active: bool,
62   input_tokens: Vec<Pubkey>,
63   input_decimals: Vec<u16>,
64   input_percentages: Vec<u16>,
65   input_price_feeds: Vec<Pubkey>,
66   fee_percent: u16,
67   total_minted_limit: u64,
68   per_period_minted_limit: u64,
69 ) -> Result<()> {
70
71   if input_tokens.len() != input_decimals.len() {
72     return Err(ErrorCode::InvalidInput.into());
73   }
74   if input_tokens.len() != input_percentages.len() {
75     return Err(ErrorCode::InvalidInput.into());
76   }
77   if input_tokens.len() != input_price_feeds.len() {
78     return Err(ErrorCode::InvalidInput.into());
79   }
80   let percentage: u16 = input_percentages.iter().sum();
81   if percentage != 10000 {
82     return Err(ErrorCode::InvalidInput.into());
```

```
83    }
84    if fee_percent > 10000 {
85        return Err(ErrorCode::InvalidInput.into());
86    }
87
88    let minter = &mut ctx.accounts.minter;
89    minter.is_active = is_active;
90    minter.input_tokens = input_tokens;
91    minter.input_decimals = input_decimals;
92    minter.input_percentages = input_percentages;
93    minter.input_price_feeds = input_price_feeds;
94    minter.fee_percent = fee_percent;
95    minter.total_minted_limit = total_minted_limit;
96    minter.per_period_minted_limit = per_period_minted_limit;
97
98    Ok(())
99  }
```

lib.rs

```
114 #[access_control(is_root(*ctx.accounts.root.key))]
115 pub fn set_burner(
116     ctx: Context<SetBurnerContext>,
117     is_active: bool,
118     output_token: Pubkey,
119     output_decimals: u16,
120     output_price_feed: Pubkey,
121     fee_percent: u16,
122     total_burned_limit: u64,
123     per_period_burned_limit: u64,
124 ) -> Result<()> {
125
126     if fee_percent > 10000 {
127         return Err(ErrorCode::InvalidInput.into());
128     }
129
130     let burner = &mut ctx.accounts.burner;
131     burner.is_active = is_active;
132     burner.output_token = output_token;
133     burner.output_decimals = output_decimals;
134     burner.output_price_feed = output_price_feed;
135     burner.fee_percent = fee_percent;
136     burner.total_burned_limit = total_burned_limit;
137     burner.per_period_burned_limit = per_period_burned_limit;
138
139     Ok(())
140 }
```

5.8.2. Remediation

Inspex suggests adding input validation to ensure that the input fee does not exceed the possible maximum fee cap by declaring the constant variable max fee for an individual state, the potential value will be defined according to the Coin98 business model. For example:

constant.rs

```
#[cfg(feature = "localhost")]
pub const ROOT_KEYS: &[&str] = &[
  "8ST8fTBGKaVPx4f1KG1zMMw4EJmSJBW2UgX1JR2pPoVa",
];

#[cfg(not(feature = "localhost"))]
pub const ROOT_KEYS: &[&str] = &[
  "EZuvvbVWibGSQpU4urZixQho2hDWtarC9bhT5NVKFpw8",
  "5UrM9csUEDBeBqMZTuuZyHRNhbRW4vQ1MgKJDrKU1U2v",
  "GnzQDYm2gvwZ8wRVmuwVAeHx5T44ovC735vDgSNhumzQ",
];

pub const APP_DATA_SEED_1: &[u8] = &[144, 146, 13, 147, 226, 199, 230, 50];
pub const APP_DATA_SEED_2: &[u8] = &[15, 81, 173, 106, 105, 203, 253, 99];
pub const ROOT_SIGNER_SEED_1: &[u8] = &[2, 151, 229, 53, 244, 77, 229, 7];
pub const ROOT_SIGNER_SEED_2: &[u8] = &[68, 203, 0, 94, 226, 230, 93, 156];

pub const MINT_FEE_CAP: u16 = 500;
pub const BURN_FEE_CAP: u16 = 500;
```

lib.rs

```
#[access_control(is_root(*ctx.accounts.root.key))]
pub fn set_minter(
  ctx: Context<SetMinterContext>,
  is_active: bool,
  input_tokens: Vec<Pubkey>,
  input_decimals: Vec<u16>,
  input_percentages: Vec<u16>,
  input_price_feeds: Vec<Pubkey>,
  fee_percent: u16,
  total_minted_limit: u64,
  per_period_minted_limit: u64,
) -> Result<()> {

  if input_tokens.len() != input_decimals.len() {
    return Err(ErrorCode::InvalidInput.into());
  }
  if input_tokens.len() != input_percentages.len() {
    return Err(ErrorCode::InvalidInput.into());
  }
```

```
77    if input_tokens.len() != input_price_feeds.len() {
78      return Err(ErrorCode::InvalidInput.into());
79    }
80    let percentage: u16 = input_percentages.iter().sum();
81    if percentage != 10000 {
82      return Err(ErrorCode::InvalidInput.into());
83    }
84    if fee_percent > MINT_FEE_CAP {
85      return Err(ErrorCode::InvalidInput.into());
86    }
87
88    let minter = &mut ctx.accounts.minter;
89    minter.is_active = is_active;
90    minter.input_tokens = input_tokens;
91    minter.input_decimals = input_decimals;
92    minter.input_percentages = input_percentages;
93    minter.input_price_feeds = input_price_feeds;
94    minter.fee_percent = fee_percent;
95    minter.total_minted_limit = total_minted_limit;
96    minter.per_period_minted_limit = per_period_minted_limit;
97
98    Ok(())
99  }
```

lib.rs

```
114 #[access_control(is_root(*ctx.accounts.root.key))]
115 pub fn set_burner(
116   ctx: Context<SetBurnerContext>,
117   is_active: bool,
118   output_token: Pubkey,
119   output_decimals: u16,
120   output_price_feed: Pubkey,
121   fee_percent: u16,
122   total_burned_limit: u64,
123   per_period_burned_limit: u64,
124 ) -> Result<()> {
125
126   if fee_percent > BURN_FEE_CAP {
127     return Err(ErrorCode::InvalidInput.into());
128   }
129
130   let burner = &mut ctx.accounts.burner;
131   burner.is_active = is_active;
132   burner.output_token = output_token;
133   burner.output_decimals = output_decimals;
134   burner.output_price_feed = output_price_feed;
135   burner.fee_percent = fee_percent;
```

```
136     burner.total_burned_limit = total_burned_limit;
137     burner.per_period_burned_limit = per_period_burned_limit;
138
139     Ok(())
140 }
```

5.9. Insufficient Logging for Privileged Functions

ID	IDX-008
Target	coin98_dollar_mint_burn
Category	General Smart Contract Vulnerability
CWE	CWE-778: Insufficient Logging
Risk	**Severity:** Very Low **Impact:** Low Privileged functions' executions cannot be monitored easily by the users. **Likelihood:** Low It is unlikely that the execution of the privileged functions will be a malicious action.
Status	Resolved The Coin98 team has resolved this issue by emitting the suggested event for the execution of privileged functions. This issue has been resolved in commit `9e3c086ed1071521fa0624d7900464b117635073`.

5.9.1. 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 privileged functions with insufficient logging are as follows:

File	Program	Function
/programs/coin98_dollar_mint_burn/src/lib.rs (L:42)	coin98_dollar_mint_burn	create_minter()
/programs/coin98_dollar_mint_burn/src/lib.rs (L:59)	coin98_dollar_mint_burn	set_minter()
/programs/coin98_dollar_mint_burn/src/lib.rs (L:102)	coin98_dollar_mint_burn	create_burner()
/programs/coin98_dollar_mint_burn/src/lib.rs (L:115)	coin98_dollar_mint_burn	set_burner()
/programs/coin98_dollar_mint_burn/src/lib.rs (L:342)	coin98_dollar_mint_burn	withdraw_token()
/programs/coin98_dollar_mint_burn/src/lib.rs (L:370)	coin98_dollar_mint_burn	unlock_token_mint()

/programs/coin98_dollar_mint_burn/src/lib.rs (L:416)	coin98_dollar_mint_burn	set_app_data()

5.9.2. Remediation

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

lib.rs

```
41  #[access_control(is_root(*ctx.accounts.root.key))]
42  pub fn create_minter(
43    ctx: Context<CreateMinterContext>,
44    _derivation_path: Vec<u8>,
45  ) -> Result<()> {
46
47    let minter = &mut ctx.accounts.minter;
48    minter.nonce = *ctx.bumps.get("minter").unwrap();
49    minter.is_active = false;
50    minter.input_tokens = Vec::new();
51    minter.input_decimals = Vec::new();
52    minter.input_percentages = Vec::new();
53    minter.input_price_feeds = Vec::new();
54
55    emit!(CreateMinterEvent{
56        _derivation_path,
57    });
58
59    Ok(())
60  }
```

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