BLOCKSEC

Security Audit

Report for UR Topup Contract Solana

Date: April 9, 2026 **Version:** 1.0

Contact: contact@blocksec.com

Contents

Report Manifest

Item	Description
Client	Mantle
Target	UR Topup Contract Solana

Version History

Version	Date	Description
1.0	April 9, 2026	First release

Signature

Digitally signed by: BlockSec (https://blocksec.com)

SHA1 digest of public key: BAB106D3708BC6DE0963CE05E9478C24B326AE1C

Date: 2026-04-10 02:03:05+0000

About BlockSec BlockSec focuses on the security of the blockchain ecosystem and collaborates with leading DeFi projects to secure their products. BlockSec is founded by top-notch security researchers and experienced experts from both academia and industry. They have published multiple blockchain security papers in prestigious conferences, reported several zero-day attacks of DeFi applications, and successfully protected digital assets that are worth more than 14 million dollars by blocking multiple attacks. They can be reached at Email, Twitter and Medium.

Chapter 1 Introduction

1.1 About the Audit Target

Information	Description
Type	Smart Contract
Language	Rust
Approach	Semi-automatic and manual verification

The audit target (hereinafter referred to as the Target) of this audit is the code repository [1] of UR Topup Contract Solana of Mantle.

This Solana program functions as an Omnichain Application (OApp) that bridges token deposits with decentralized exchange capabilities. It integrates the Jupiter Aggregator, allowing users to swap arbitrary tokens into USDC before initiating cross-chain transfers via the LayerZero protocol. The contract supports two primary deposit flows: a standard model where transaction fees are deducted from the user's swap proceeds, and a sponsored model where a third-party payer covers the fee. It uses Cross-Program Invocations (CPI) to seamlessly coordinate swaps and cross-chain transfers.

Note this audit only focuses on the smart contracts in the following directories/files:

- programs/my_oapp

Other files are not within the scope of the audit. Additionally, all dependencies of the Target are considered reliable in terms of both functionality and security, and are therefore not included in the audit scope.

The auditing process is iterative. Specifically, we would audit the commits that fix the discovered issues. If there are new issues, we will continue this process. The commit SHA values during the audit are shown in the following table. Our audit report is responsible for the code in the initial version (`Version 1`), as well as new code (in the following versions) to fix issues in the audit report. Code prior to and including the baseline version (`Version 0`), where applicable, is outside the scope of this audit and assumes to be reliable and secure.

Project	Version	Commit Hash
UR Topup Contract Solana	`Version 1`	`2d550fd6dd32657106eb3e19e614bacb8be3c2dc`
	`Version 2`	`33bb65fff13f6bbc0bf702b1daf4fed7a1483904`

1.2 Disclaimer

This audit report does not constitute investment advice or a personal recommendation. It does not consider, and should not be interpreted as considering or having any bearing on, the potential economics of a token, token sale or any other product, service or other asset. Any entity should not rely on this report in any way, including for the purpose of making any decisions to buy or sell any token, product, service or other asset.

[1] https://github.com/mantle-xyz/ur-topup-contract-sol

This audit report is not an endorsement of any particular project or team, and the report does not guarantee the security of any particular project. This audit does not give any warranties on discovering all security issues of the Target, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit cannot be considered comprehensive, we always recommend proceeding with independent audits and a public bug bounty program to ensure the security of the Target.

The scope of this audit is limited to the code mentioned in Section 1.1. Unless explicitly specified, the security of the language itself (e.g., the solidity language), the underlying compiling toolchain and the computing infrastructure are out of the scope.

1.3 Procedure of Auditing

We perform the audit according to the following procedure.

- **Vulnerability Detection** We first scan the Target with automatic code analyzers, and then manually verify (reject or confirm) the issues reported by them.
- **Semantic Analysis** We study the business logic of the Target and conduct further investigation on the possible vulnerabilities using an automatic fuzzing tool (developed by our research team). We also manually analyze possible attack scenarios with independent auditors to cross-check the result.
- **Recommendation** We provide some useful advice to developers from the perspective of good programming practice, including gas optimization, code style, and etc.

We show the main concrete checkpoints in the following.

1.3.1 Security Issues

* Access control
* Permission management
* Whitelist and blacklist mechanisms
* Initialization consistency
* Improper use of the proxy system
* Reentrancy
* Denial of Service (DoS)
* Untrusted external call and control flow
* Exception handling
* Data handling and flow
* Events operation
* Error-prone randomness
* Oracle security
* Business logic correctness
* Semantic and functional consistency
* Emergency mechanism
* Economic and incentive impact

1.3.2 Additional Recommendation

* Gas optimization
* Code quality and style

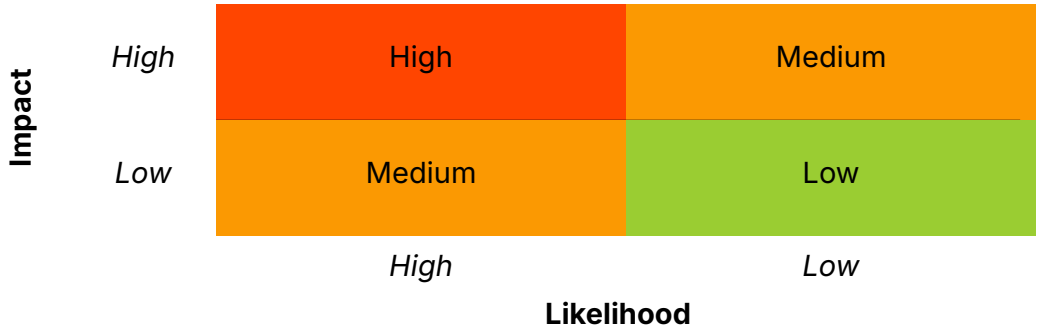
Note *The previous checkpoints are the main ones. We may use more checkpoints during the auditing process according to the functionality of the project.*

1.4 Security Model

To evaluate the risk, we follow the standards or suggestions that are widely adopted by both industry and academy, including OWASP Risk Rating Methodology [2] and Common Weakness Enumeration [3]. The overall *severity* of the risk is determined by *likelihood* and *impact*. Specifically, likelihood is used to estimate how likely a particular vulnerability can be uncovered and exploited by an attacker, while impact is used to measure the consequences of a successful exploit.

In this report, both likelihood and impact are categorized into two ratings, i.e., *high* and *low* respectively, and their combinations are shown in Table 1.1.

Table 1.1: Vulnerability Severity Classification



Accordingly, the severity measured in this report are classified into three categories: **High**, **Medium**, **Low**. For the sake of completeness, **Undetermined** is also used to cover circumstances when the risk cannot be well determined.

Furthermore, the status of a discovered item will fall into one of the following five categories:

- **Undetermined** No response yet.
- **Acknowledged** The item has been received by the client, but not confirmed yet.
- **Confirmed** The item has been recognized by the client, but not fixed yet.
- **Partially Fixed** The item has been confirmed and partially fixed by the client.
- **Fixed** The item has been confirmed and fixed by the client.

[2] https://owasp.org/www-community/OWASP_Risk_Rating_Methodology

[3] https://cwe.mitre.org/

Chapter 2 Findings

In total, we found **eight** potential security issues. Besides, we have **four** recommendations and **two** notes.

- High Risk: 4
- Medium Risk: 1
- Low Risk: 3
- Recommendation: 4
- Note: 2

ID	Severity	Description	Category	Status
1	High	Shared `remaining_accounts` causes conflict between Jupiter and LayerZero	Security Issue	Fixed
2	High	Insufficient account space allocation for dynamic data	Security Issue	Fixed
3	High	Malicious constructed Jupiter `swap_data` and `remaining_accounts` might cause `payer` to lose funds	Security Issue	Fixed
4	High	Unvalidated `evm_output_token` in cross chain messages will cause users to lose funds	Security Issue	Fixed
5	Medium	Endpoint delegate is not synchronized when `store.admin` changes	Security Issue	Fixed
6	Low	User-controlled `fee_amount_via_usdc` enables fee circumvention	Security Issue	Fixed
7	Low	Lack of validation for `user_input_token_account.mint` versus swap input token	Security Issue	Fixed
8	Low	Lack of option type check in the function `init_store()`	Security Issue	Fixed
9	-	Non zero address checks	Recommendation	Fixed
10	-	Revise constant variable naming typos	Recommendation	Fixed
11	-	Add non zero value check for `usdc_received`	Recommendation	Fixed
12	-	Lack of check in function `set_usdc_mint()`	Recommendation	Confirmed
13	-	`Payer` should perform off-chain validation before signing	Note	-
14	-	Potential centralization risks	Note	-

The details are provided in the following sections.

2.1 Security Issue

2.1.1 Shared `remaining_accounts` causes conflict between Jupiter and LayerZero

Severity High

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The functions `deposit_via_aggregator()` and `deposit_via_aggregator_sponsored()` are used to process a swap on Jupiter or send a cross-chain message to LayerZero. However, they use the same `ctx.remaining_accounts` for two different CPIs: Jupiter swap CPI in the function `execute_jupiter_swap()` and the function `build_and_send_lz_message()`. These two CPIs require different `ctx.remaining_accounts`, but the program passes the same `ctx.remaining_accounts` to both, which creates a conflict. As a result, this might cause DoS or unexpected behaviors.

```
211  impl<'info> DepositViaAggregator<'info> {
212      pub fn apply(
213              ctx: &mut Context<'_, '_, '_, 'info, DepositViaAggregator<'info>>,
214              params: &DepositViaAggregatorParams,
215          ) -> Result<u64> {
216      }
217  }
```

Listing 2.1: programs/my_oapp/src/instructions/deposit_via_aggregator.rs

```
389  impl<'info> DepositViaAggregatorSponsored<'info> {
390      pub fn apply(
391              ctx: &mut Context<'_, '_, '_, 'info, DepositViaAggregatorSponsored<'info>>,
392              params: &DepositViaAggregatorSponsoredParams,
393          ) -> Result<u64> {
394      }
395  }
```

Listing 2.2: programs/my_oapp/src/instructions/deposit_via_aggregator.rs

```
38  /// Execute Jupiter swap via CPI
39  /// The swap_data is the serialized instruction data from Jupiter API
40  fn execute_jupiter_swap<'a>(
41      jupiter_program: &AccountInfo<'a>,
42      user: &Signer<'a>,
43      remaining_accounts: &[AccountInfo<'a>],
44      swap_data: &[u8],
45  ) -> Result<()> {
46      let account_metas: Vec<AccountMeta> = remaining_accounts
47          .iter()
48          .map(|acc| {
49              if acc.is_writable {
50                  AccountMeta::new(*acc.key, acc.is_signer)
51              } else {
52                  AccountMeta::new_readonly(*acc.key, acc.is_signer)
```

```
53              }
54          })
55          .collect();
56
57      let jupiter_ix = Instruction {
58          program_id: *jupiter_program.key,
59          accounts: account_metas,
60          data: swap_data.to_vec(),
61      };
62
63      let mut account_infos = vec![jupiter_program.clone(), user.to_account_info()];
64      account_infos.extend(remaining_accounts.iter().cloned());
65
66      invoke(&jupiter_ix, &account_infos)?;
67      Ok(())
68  }
```

Listing 2.3: programs/my_oapp/src/instructions/deposit_via_aggregator.rs

```
102 /// Build cross-chain message, prepare LZ options, and send via LayerZero endpoint
103 fn build_and_send_lz_message<'info>(
104     evm_recipient: &[u8; 20],
105     input_token_mint: Pubkey,
106     input_amount: u64,
107     usdc_amount: u64,
108     evm_output_token: &[u8; 20],
109     native_fee: u64,
110     store: &Account<'info, Store>,
111     peer: &Account<'info, PeerConfig>,
112     remaining_accounts: &[AccountInfo<'info>],
113 ) -> Result<()> {
114     let deposit_msg = deposit_codec::DepositMessage::new(
115         pad_evm_address(evm_recipient),
116         input_token_mint.to_bytes(),
117         input_amount,
118         usdc_amount,
119         pad_evm_address(evm_output_token),
120     );
121     let message = deposit_codec::encode(&deposit_msg);
122
123     let seeds: &[&[u8]] = &[STORE_SEED, &[store.bump]];
124
125     let default_options = if store.default_lz_options.is_empty() {
126         default_lz_options()
127     } else {
128         store.default_lz_options.clone()
129     };
130     let options = peer
131         .enforced_options
132         .combine_options(&None::<Vec<u8>>, &default_options)?;
133
134     let send_params = SendParams {
135         dst_eid: store.dst_eid,
```

```
136             receiver: peer.peer_address,
137             message,
138             options,
139             native_fee,
140             lz_token_fee: 0,
141         };
142
143         oapp::endpoint_cpi::send(
144             ENDPOINT_ID,
145             store.key(),
146             remaining_accounts,
147             seeds,
148             send_params,
149         )?;
150
151         Ok(())
152     }
```

Listing 2.4: programs/my_oapp/src/instructions/deposit_via_aggregator.rs

Impact As a result, this might cause DoS or unexpected behaviors.

Suggestion Revise the logic accordingly.

2.1.2 Insufficient account space allocation for dynamic data

Severity High

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The `PeerConfig` account uses `std::mem::size_of::<Self>()` to calculate the required space for account initialization. In Rust language, `std::mem::size_of` only measures the fixed-size "header" (metadata) of a `Vec`, which is typically 24 bytes representing the pointer, length, and capacity. It does not account for the heap-allocated capacity reserved for the dynamic data itself.

Because the `EnforcedOptions` struct contains `Vec` fields annotated with `#[max_len(...)]`, the actual data required for these fields exceeds the size calculated by `std::mem::size_of`. Consequently, the function `set_peer_config()` creates accounts that are significantly under-allocated, which will cause transaction failure (AccountDataTooSmall) when attempting to store data in those vectors.

```
14  #[account(
15      init_if_needed,
16      payer = admin,
17      space = PeerConfig::SIZE,
18      seeds = [PEER_SEED, &store.key().to_bytes(), &params.remote_eid.to_be_bytes()],
19      bump
20  )]
21  /// Peer configuration PDA for a specific remote chain
22  pub peer: Account<'info, PeerConfig>,
```

Listing 2.5: programs/my_oapp/src/instructions/set_peer_config.rs

```rust
#[account]
pub struct PeerConfig {
    pub peer_address: [u8; 32],
    pub enforced_options: EnforcedOptions,
    pub bump: u8,
}

impl PeerConfig {
    pub const SIZE: usize = 8 + std::mem::size_of::<Self>();
}

#[derive(Clone, Default, AnchorSerialize, AnchorDeserialize, InitSpace)]
pub struct EnforcedOptions {
    #[max_len(ENFORCED_OPTIONS_SEND_MAX_LEN)]
    pub send: Vec<u8>,
    #[max_len(ENFORCED_OPTIONS_SEND_AND_CALL_MAX_LEN)]
    pub send_and_call: Vec<u8>,
}
```

Listing 2.6: programs/my_oapp/src/state/peer_config.rs

Impact The function `set_peer_config()` creates accounts that are significantly under-allocated, which will cause transaction failure (`AccountDataTooSmall`) when attempting to store data in those vectors.

Suggestion Revise the logic accordingly.

2.1.3 Malicious constructed Jupiter `swap_data` and `remaining_accounts` might cause `payer` to lose funds

Severity High

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The function `execute_jupiter_swap()` builds and invokes a Jupiter instruction with user-controlled `params.swap_data` and `ctx.remaining_accounts`. However, there is no validation to make sure that the input token is from the `user` rather than `payer`. Since, in sponsored flow, both `user` and `payer` are transaction signers, if `ctx.remaining_accounts` binds Jupiter authority to the `payer` token account, malicious constructed Jupiter `swap_data` and `remaining_accounts` might cause swap execution to steal the `payer`'s funds.

```rust
/// Execute Jupiter swap via CPI
/// The swap_data is the serialized instruction data from Jupiter API
fn execute_jupiter_swap<'a>(
    jupiter_program: &AccountInfo<'a>,
    user: &Signer<'a>,
    remaining_accounts: &[AccountInfo<'a>],
    swap_data: &[u8],
) -> Result<()> {
    let account_metas: Vec<AccountMeta> = remaining_accounts
        .iter()
```

```
48          .map(|acc| {
49              if acc.is_writable {
50                  AccountMeta::new(*acc.key, acc.is_signer)
51              } else {
52                  AccountMeta::new_readonly(*acc.key, acc.is_signer)
53              }
54          })
55          .collect();
56
57      let jupiter_ix = Instruction {
58          program_id: *jupiter_program.key,
59          accounts: account_metas,
60          data: swap_data.to_vec(),
61      };
62
63      let mut account_infos = vec![jupiter_program.clone(), user.to_account_info()];
64      account_infos.extend(remaining_accounts.iter().cloned());
65
66      invoke(&jupiter_ix, &account_infos)?;
67      Ok(())
68  }
```

Listing 2.7: programs/my_oapp/src/instructions/deposit_via_aggregator.rs

Impact If `ctx.remaining_accounts` binds Jupiter authority to the `payer` token account, malicious constructed Jupiter `swap_data` and `remaining_accounts` might cause swap execution to steal the `payer`'s funds.

Suggestion Revise the logic accordingly.

2.1.4 Unvalidated `evm_output_token` in cross chain messages will cause users to lose funds

Severity High

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the functions `deposit_usdc()`, `deposit_via_aggregator()`, and `deposit_via_aggregator_sponsored()`, the function `send()` is invoked with `send_params`, which contains the field `evm_output_token`. Under our assumptions, the field `evm_output_token` represents the fiat tokens to be sent to users in the destination chain. If the field `evm_output_token` is an invalid token, users' funds might be lost. This is because the users' deposited USDC in the source chain will not be refunded and the destination chain will not send the fiat tokens to users.

```
80      ctx.accounts.store.default_lz_options = if params.default_lz_options.is_empty() {
81          default_lz_options()
82      } else {
83          params.default_lz_options.clone()
84      };
```

Listing 2.8: programs/my_oapp/src/instructions/init_store.rs

```rust
pub fn combine_options(
    &self,
    compose_msg: &Option<Vec<u8>>,
    extra_options: &Vec<u8>,
) -> Result<Vec<u8>> {
    let enforced_options = self.get_enforced_options(compose_msg);
    oapp::options::combine_options(enforced_options, extra_options)
}
```

Listing 2.9: programs/my_oapp/src/state/peer_config.rs

Impact If the field `evm_output_token` is an invalid token, users' funds might be lost.

Suggestion Add checks to `evm_output_token`.

2.1.5 Endpoint delegate is not synchronized when `store.admin` changes

Severity Medium

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The endpoint delegate is set during `InitStore` through `register_oapp(delegate = store.admin)`. However, the function `accept_admin()` updates `store.admin` and clears `pending_admin`, but does not synchronize LayerZero Endpoint OApp delegate state.

This creates authority conflict between local governance (`store.admin`) and endpoint-side delegate authorization.

```rust
    // Register OApp with LayerZero Endpoint (only on first-time init).
    // If remaining_accounts is empty, skip registration -the OApp is
    // already registered (e.g. re-initializing after close_store).
    if !ctx.remaining_accounts.is_empty() {
        let register_params = RegisterOAppParams { delegate: ctx.accounts.store.admin };
        let seeds: &[&[u8]] = &[STORE_SEED, &[ctx.accounts.store.bump]];
        oapp::endpoint_cpi::register_oapp(
            ENDPOINT_ID,
            ctx.accounts.store.key(),
            ctx.remaining_accounts,
            seeds,
            register_params,
        )?;
    }
```

Listing 2.10: programs/my_oapp/src/instructions/init_store.rs

```rust
impl AcceptAdmin<'_> {
    pub fn apply(ctx: &mut Context<AcceptAdmin>) -> Result<()> {
        let store = &mut ctx.accounts.store;
        let old_admin = store.admin;

        store.admin = store.pending_admin;
        store.pending_admin = Pubkey::default();

```

```
65          emit!(AdminTransferred {
66              old_admin,
67              new_admin: store.admin,
68          });
69
70          Ok(())
71      }
72  }
```

Listing 2.11: programs/my_oapp/src/instructions/transfer_admin.rs

Impact This creates authority conflict between local governance (`store.admin`) and endpoint-side delegate authorization.

Suggestion Revise the admin transfer logic accordingly.

2.1.6 User-controlled `fee_amount_via_usdc` enables fee circumvention

Severity Low

Status Fixed in `Version 2`

Introduced by `Version 1`

Description Users can initiate a cross-chain transaction through the function `deposit_via_aggregator_sponsored()`. Within this function, the `fee_amount` is obtained from `params.fee_amount_via_usdc`. This design allows a user to set `fee_amount_via_usdc` to be 0, therefore circumventing the fee charge.

```
411      // Step 2: Payer pays the fee (from payer's USDC, NOT user's)
412      let fee_amount = params.fee_amount_via_usdc;
413      if fee_amount > 0 {
414          token::transfer(
415              CpiContext::new(
416                  ctx.accounts.token_program.to_account_info(),
417                  Transfer {
418                      from: ctx.accounts.payer_usdc_account.to_account_info(),
419                      to: ctx.accounts.fee_usdc_account.to_account_info(),
420                      authority: ctx.accounts.payer.to_account_info(),
421                  },
422              ),
423              fee_amount,
424          )?;
425      }
```

Listing 2.12: programs/my_oapp/src/instructions/deposit_via_aggregator.rs

Impact This design allows a user to set `fee_amount_via_usdc` to 0, thereby circumventing the fee charge.

Suggestion Revise the logic, that `fee_amount` is obtained from the `store` account.

2.1.7 Lack of validation for `user_input_token_account.mint` versus swap input token

Severity Low

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the file `deposit_via_aggregator.rs`, the function `resolve_usdc_received()` performs a swap using `ctx.remaining_accounts` while the variable `input_token_mint` is derived solely from the variable `user_input_token_account.mint` and written into the cross-chain message. However, the program does not validate that `user_input_token_account.mint` matches the actual input token implied by the swap accounts in `ctx.remaining_accounts`. As a result, this flaw might result in incorrect cross-chain messages.

```
217     let input_token_mint = ctx.accounts.user_input_token_account.mint;
218
219     require!(params.input_amount > 0, MyOAppError::AmountZero);
220
221     // Step 1: Swap or skip
222     let usdc_received = resolve_usdc_received(
223         &params.swap_data,
224         params.input_amount,
225         &ctx.accounts.jupiter_program,
226         &ctx.accounts.user,
227         &mut ctx.accounts.user_usdc_account,
228         ctx.remaining_accounts,
229     )?;
```

Listing 2.13: programs/my_oapp/src/instructions/deposit_via_aggregator.rs

Impact This flaw might result in incorrect cross-chain messages.

Suggestion Revise the logic accordingly.

2.1.8 Lack of option type check in the function `init_store()`

Severity Low

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The function `init_store()` does not check that the option type must be 3. However the function `combine_options()`, which is invoked by the functions `build_and_send_lz_message()` and `deposit_usdc()`, requires the option type to be 3. If the option type is mistakenly set, the program's main functionalities will be unavailable.

```
80      ctx.accounts.store.default_lz_options = if params.default_lz_options.is_empty() {
81          default_lz_options()
82      } else {
83          params.default_lz_options.clone()
84      };
```

Listing 2.14: programs/my_oapp/src/instructions/init_store.rs

```rust
pub fn combine_options(
    &self,
    compose_msg: &Option<Vec<u8>>,
    extra_options: &Vec<u8>,
) -> Result<Vec<u8>> {
    let enforced_options = self.get_enforced_options(compose_msg);
    oapp::options::combine_options(enforced_options, extra_options)
}
```

Listing 2.15: programs/my_oapp/src/state/peer_config.rs

Impact If the option type is mistakenly set, the program's main functionalities will be unavailable.

Suggestion Add option type check in the function `init_store()`.

2.2 Recommendation

2.2.1 Non zero address checks

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In functions `deposit_usdc()` and `deposit_via_aggregator()`, several address variables (e.g., `evm_recipient`) are not checked to ensure they are not zero. It is recommended to add such checks to prevent potential misoperations.

```rust
pub fn apply(ctx: &mut Context<DepositUsdc>, params: &DepositUsdcParams) -> Result<u64> {
    let store = &ctx.accounts.store;

    // Validate amount
    require!(params.usdc_amount > 0, MyOAppError::AmountZero);

    // Use fixed fee amount from store
    let fee_amount = store.fee_amount;

    // Calculate net amount after fee
    let net_amount = params.usdc_amount
        .checked_sub(fee_amount)
        .ok_or(MyOAppError::InsufficientAmount)?;

    // Ensure net amount is positive
    require!(net_amount > 0, MyOAppError::InsufficientAmount);

    require!(net_amount >= params.min_usdc_amount, MyOAppError::SlippageExceeded);

    // Transfer fee to fee receiver (if any)
    if fee_amount > 0 {
        token::transfer(
            CpiContext::new(
                ctx.accounts.token_program.to_account_info(),
                Transfer {
```

```
                from: ctx.accounts.user_usdc_account.to_account_info(),
                to: ctx.accounts.fee_usdc_account.to_account_info(),
                authority: ctx.accounts.user.to_account_info(),
            },
        ),
        fee_amount,
    )?;
}

// Transfer net USDC to deposit address
token::transfer(
    CpiContext::new(
        ctx.accounts.token_program.to_account_info(),
        Transfer {
            from: ctx.accounts.user_usdc_account.to_account_info(),
            to: ctx.accounts.deposit_usdc_account.to_account_info(),
            authority: ctx.accounts.user.to_account_info(),
        },
    ),
    net_amount,
)?;

// Prepare and send LayerZero message
let deposit_msg = deposit_codec::DepositMessage::new_usdc_deposit(
    params.evm_recipient,
    store.usdc_mint.to_bytes(),
    net_amount,
    params.evm_output_token,
);
let message = deposit_codec::encode(&deposit_msg);

// Prepare seeds for Store PDA signing
let seeds: &[&[u8]] = &[STORE_SEED, &[store.bump]];

// Use default LZ options from Store, combined with enforced options
let default_options = if store.default_lz_options.is_empty() {
    default_lz_options()
} else {
    store.default_lz_options.clone()
};
let options = ctx
    .accounts
    .peer
    .enforced_options
    .combine_options(&None::<Vec<u8>>, &default_options)?;

// Prepare SendParams for LayerZero endpoint (using store.dst_eid)
let send_params = SendParams {
    dst_eid: store.dst_eid,
    receiver: ctx.accounts.peer.peer_address,
    message,
    options,
    native_fee: params.native_fee,
```

```
149             lz_token_fee: 0,
150         };
151
152         // Send cross-chain message via LayerZero
153         oapp::endpoint_cpi::send(
154             ENDPOINT_ID,
155             ctx.accounts.store.key(),
156             ctx.remaining_accounts,
157             seeds,
158             send_params,
159         )?;
160
161         // Emit deposit event
162         emit!(DepositUsdcEvent {
163             user: ctx.accounts.user.key(),
164             usdc_amount: params.usdc_amount,
165             fee_amount,
166             net_amount,
167             dst_eid: store.dst_eid,
168             evm_recipient: params.evm_recipient,
169             evm_output_token: params.evm_output_token,
170         });
171
172         Ok(net_amount)
173     }
```

Listing 2.16: programs/my_oapp/src/instructions/deposit_usdc.rs

```
212     pub fn apply(
213         ctx: &mut Context<'_, '_, '_, 'info, DepositViaAggregator<'info>>,
214         params: &DepositViaAggregatorParams,
215     ) -> Result<u64> {
216         let store = &ctx.accounts.store;
217         let input_token_mint = ctx.accounts.user_input_token_account.mint;
218
219         require!(params.input_amount > 0, MyOAppError::AmountZero);
220
221         // Step 1: Swap or skip
222         let usdc_received = resolve_usdc_received(
223             &params.swap_data,
224             params.input_amount,
225             &ctx.accounts.jupiter_program,
226             &ctx.accounts.user,
227             &mut ctx.accounts.user_usdc_account,
228             ctx.remaining_accounts,
229         )?;
230
231         require!(usdc_received >= params.min_usdc_amount, MyOAppError::SlippageExceeded);
232
233         // Step 2: Deduct fee from user
234         let fee_amount = store.fee_amount;
235         let net_amount = usdc_received
236             .checked_sub(fee_amount)
```

```
237             .ok_or(MyOAppError::InsufficientAmount)?;
238         require!(net_amount > 0, MyOAppError::InsufficientAmount);
239
240         if fee_amount > 0 {
241             token::transfer(
242                 CpiContext::new(
243                     ctx.accounts.token_program.to_account_info(),
244                     Transfer {
245                         from: ctx.accounts.user_usdc_account.to_account_info(),
246                         to: ctx.accounts.fee_usdc_account.to_account_info(),
247                         authority: ctx.accounts.user.to_account_info(),
248                     },
249                 ),
250                 fee_amount,
251             )?;
252         }
253
254         // Step 3: Transfer net USDC to deposit
255         token::transfer(
256             CpiContext::new(
257                 ctx.accounts.token_program.to_account_info(),
258                 Transfer {
259                     from: ctx.accounts.user_usdc_account.to_account_info(),
260                     to: ctx.accounts.deposit_usdc_account.to_account_info(),
261                     authority: ctx.accounts.user.to_account_info(),
262                 },
263             ),
264             net_amount,
265         )?;
266
267         // Step 4: Send cross-chain message
268         build_and_send_lz_message(
269             &params.evm_recipient,
270             input_token_mint,
271             params.input_amount,
272             net_amount,
273             &params.evm_output_token,
274             params.native_fee,
275             &ctx.accounts.store,
276             &ctx.accounts.peer,
277             ctx.remaining_accounts,
278         )?;
279
280         emit!(DepositViaAggregatorEvent {
281             user: ctx.accounts.user.key(),
282             input_token: input_token_mint,
283             input_amount: params.input_amount,
284             usdc_received,
285             fee_amount,
286             net_amount,
287             dst_eid: store.dst_eid,
288             evm_recipient: params.evm_recipient,
289             evm_output_token: params.evm_output_token,
```

```
290         });
291
292         Ok(net_amount)
293     }
```

Listing 2.17: programs/my_oapp/src/instructions/deposit_via_aggregator.rs

Suggestion Add non-zero address checks accordingly.

2.2.2 Revise constant variable naming typos

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The code defines some hardcoded destination EID constants. In LayerZero, eid 40181 is mantle-test while eid 40246 is mantle-sepolia. It is recommended to rename constants `MANTLE_SEPOLIA_EID` and `TARGET_CHAIN_EID` to avoid confusion.

```
3   // LayerZero Endpoint IDs
4   pub const MANTLE_MAINNET_EID: u32 = 30181;
5   pub const MANTLE_SEPOLIA_EID: u32 = 40181;
6
7   // Your target chain EID
8   pub const TARGET_CHAIN_EID: u32 = 40246;
```

Listing 2.18: programs/my_oapp/src/state/store.rs

Suggestion It is recommended to rename constants `MANTLE_SEPOLIA_EID` and `TARGET_CHAIN_EID` to avoid mis-leading.

2.2.3 Add non zero value check for `usdc_received`

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the function `deposit_via_aggregator_sponsored()`, if `usdc_received` is zero, a transaction can proceed and send a message. Such unnecessary cross-chain messages may waste the `payer`'s funds. It is recommended to add a non zero check for `usdc_received` to prevent meaningless messages and save the `payer`'s funds.

```
395         let input_token_mint = ctx.accounts.user_input_token_account.mint;
396
397         require!(params.input_amount > 0, MyOAppError::AmountZero);
398
399         // Step 1: Swap or skip (reuses shared helper)
400         let usdc_received = resolve_usdc_received(
401             &params.swap_data,
402             params.input_amount,
403             &ctx.accounts.jupiter_program,
404             &ctx.accounts.user,
405             &mut ctx.accounts.user_usdc_account,
406             ctx.remaining_accounts,
```

```
407         )?;
408
409         require!(usdc_received >= params.min_usdc_amount, MyOAppError::SlippageExceeded);
410
411         // Step 2: Payer pays the fee (from payer's USDC, NOT user's)
412         let fee_amount = params.fee_amount_via_usdc;
413         if fee_amount > 0 {
414             token::transfer(
415                 CpiContext::new(
416                     ctx.accounts.token_program.to_account_info(),
417                     Transfer {
418                         from: ctx.accounts.payer_usdc_account.to_account_info(),
419                         to: ctx.accounts.fee_usdc_account.to_account_info(),
420                         authority: ctx.accounts.payer.to_account_info(),
421                     },
422                 ),
423                 fee_amount,
```

Listing 2.19: programs/my_oapp/src/instructions/deposit_via_aggregator.rs

Suggestion It is recommended to add a zero-amount check for `usdc_received` to prevent meaningless messages and save the `payer`'s funds.

2.2.4 Lack of check in function `set_usdc_mint()`

Status Confirmed

Introduced by `Version 2`

Description The `ROLE_OPERATOR_ADMIN` can invoke function `set_usdc_mint()` to update the `USDC` token mint. The function validates that the mint account has 6 decimals by checking `data[SPL_MINT_DECIMALS_OFFSET] == 6,` but does not additionally verify that the usdc_mint's owner is the `SPL Token Program`. This also applies to the function `init_store()`. In the event of a misconfiguration by the `ROLE_OPERATOR_ADMIN`, an invalid token mint could be set, causing the protocol to not function as expected.

```
73      require!(params.usdc_mint != Pubkey::default(), UrOAppError::InvalidUsdcMint);
74      require!(ctx.accounts.usdc_mint_account.key() == params.usdc_mint, UrOAppError::
            InvalidUsdcMint);
75      {
76          let data = ctx.accounts.usdc_mint_account.try_borrow_data()?;
77          require!(data.len() >= 45, UrOAppError::InvalidUsdcDecimals);
78          require!(data[44] == 6, UrOAppError::InvalidUsdcDecimals);
79      }
```

Listing 2.20: programs/ur_oapp/src/instructions/init_store.rs

```
68  pub fn apply(ctx: &mut Context<SetUsdcMint>, params: &SetUsdcMintParams) -> Result<()> {
69      require!(params.usdc_mint != Pubkey::default(), UrOAppError::InvalidUsdcMint);
70      require!(ctx.accounts.usdc_mint_account.key() == params.usdc_mint, UrOAppError::
            InvalidUsdcMint);
71      require_mint_decimals(&ctx.accounts.usdc_mint_account, EXPECTED_USDC_DECIMALS)?;
72      ctx.accounts.store.usdc_mint = params.usdc_mint;
```

```
73        emit!(UsdcMintUpdatedEvent { caller: ctx.accounts.caller.key(), usdc_mint: params.usdc_mint
              });
74        Ok(())
75    }
```

Listing 2.21: programs/ur_oapp/src/instructions/set_deposit_config.rs

Suggestion Add a check to ensure that the `usdc_mint_account`'s owner is the `SPL Token Program`.

Feedback from the project The project confirmed this finding. The correctness of the `USDC` mint is ensured off-chain, and the precision constraint is intended to prevent calculation inconsistencies caused by changes in `USDC` decimals.

2.3 Note

2.3.1 `Payer` should perform off-chain validation before signing

Introduced by `Version 1`

Description It is noted that before signing the transaction, the payer should perform appropriate off-chain validation of the parameters (e.g., checking whether `params.fee_amount_via_usdc` is reasonable) to avoid paying an excessive fee or other unintended behaviours.

2.3.2 Potential centralization risks

Introduced by `Version 1`

Description In this project, several privileged roles (e.g., the upgrade authority) can conduct sensitive operations, which introduces potential centralization risks. For example, the upgrade authority can close and re-initiate the `store`. If the private keys of the privileged accounts are lost or maliciously exploited, it could pose a significant risk to the protocol. Additionally, the `store` PDA can be recreated at the same address via functions `close_store()` and `init_store()`, while existing `RoleEntry` PDAs are not removed. As a result, previously granted roles may retain access. The project should ensure that all `RoleEntry` accounts are properly revoked or removed during the `store` PDA lifecycle, particularly before or during reinitialization.


BOOST WEB3 THROUGH NEXT-GENERATION SECURITY & USABILITY INNOVATIONS