BLOCKSEC

Security Audit
Report for Perp Contracts

Date: April 24, 2025 **Version:** 1.0
Contact: contact@blocksec.com

Contents

Report Manifest

Item	Description
Client	Pacifica Fi
Target	Perp Contracts

Version History

Version	Date	Description
1.0	April 24, 2025	First release

Signature

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

SHA1 digest of public key: BAB106D3708BC6DE0963CE05E9478C24B326AE1C

Date: 2026-04-30 13:49:25+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 Target Contracts

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

The target of this audit is the code repository Perp Contracts [1] for Pacifica Fi. The audited protocol is built using `Anchor` and functions similarly to a hot wallet. It handles user deposits and withdrawals based on logs and backend logic. Note this audit only focuses on the smart contracts located in the following directories/files:

- perp-contracts/programs/pacifica-mainnet/src/lib.rs

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.

Branch	Version	Commit Hash
Perp Contracts	`Version 1`	`b4c48da19e17efd015bc5c57a91e13731ee199e8`
	`Version 2`	`e3631ce1efe06308e838a8785c0ac71742346b4d`

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.

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 smart contracts, 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 smart contracts.

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] https://github.com/pacifica-fi/perp-contracts/

1.3 Procedure of Auditing

We perform the audit according to the following procedure.

- **Vulnerability Detection** We first scan smart contracts with automatic code analyzers, and then manually verify (reject or confirm) the issues reported by them.
- **Semantic Analysis** We study the business logic of smart contracts 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 Software Security

* Reentrancy
* DoS
* Access control
* Data handling and data flow
* Exception handling
* Untrusted external call and control flow
* Initialization consistency
* Events operation
* Error-prone randomness
* Improper use of the proxy system

1.3.2 DeFi Security

* Semantic consistency
* Functionality consistency
* Permission management
* Business logic
* Token operation
* Emergency mechanism
* Oracle security
* Whitelist and blacklist
* Economic impact
* Batch transfer

1.3.3 NFT Security

* Duplicated item
* Verification of the token receiver
* Off-chain metadata security

1.3.4 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.

Table 1.1: Vulnerability Severity Classification

Impact \ Likelihood	*High*	*Low*
High	High	Medium
Low	Medium	Low

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.

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 four 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.
- **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 **two** potential security issues. Besides, we have **four** recommendations and **two** notes.

- Low Risk: 2
- Recommendation: 4
- Note: 2

ID	Severity	Description	Category	Status
1	Low	Potential DoS in function `batch_withdraw()`	DeFi Security	Confirmed
2	Low	Lack of minimum value check in function `deposit()`	DeFi Security	Confirmed
3	-	Redundant checks and unused error codes	Recommendation	Confirmed
4	-	Lack of explicit error messages for access control violations	Recommendation	Confirmed
5	-	Lack of nonce check in function `deposit()`	Recommendation	Confirmed
6	-	Lack of state check in functions `pause()` and `unpause()`	Recommendation	Confirmed
7	-	Potential centralization risk	Note	-
8	-	Potential risk of front-running in contract initialization	Note	-

The details are provided in the following sections.

2.1 DeFi Security

2.1.1 Potential DoS in function `batch_withdraw()`

Severity Low

Status Confirmed

Introduced by `Version 1`

Description The `batch_withdraw()` function can be invoked by a privileged role, `withdraw_authority`, to process multiple user withdrawals in a single transaction by transferring `USDC` to each user's designated token account. However, a `for` loop is used to iterate through each withdrawal request, performing checks such as verifying that the provided token account matches the intended user. If the frontend fails to pre-check the input properly, a malicious user could craft a withdrawal request that includes invalid data early in the array, causing the entire transaction to revert. This could be abused to deny service to other legitimate users whose withdrawals are included in the same batch.

```
65    pub fn batch_withdraw<'a, 'b, 'c, 'info>(
66        ctx: Context<'a, 'b, 'c, 'info, BatchWithdraw<'info>>,
67        amounts: Vec<UserWithdraw>,
```

```
68          nonce: u64
69      ) -> Result<()> {
70          require!(!ctx.accounts.central_state.paused, ErrorCode::CentralStatePaused);
71          require!(amounts.len() == ctx.remaining_accounts.len(), ErrorCode::MismatchingLengths);
72          require!(nonce == ctx.accounts.central_state.withdraw_nonce, ErrorCode::InvalidNonce);
73
74
75          let signer_seeds = &[
76              b"central_state".as_ref(),
77              &[ctx.bumps.central_state]
78          ];
79          let signer = &[&signer_seeds[..]];
80
81
82          for (i, recipient_token_account) in ctx.remaining_accounts.iter().enumerate() {
83
84
85              // Validate that the token account is the correct ATA for the user
86              let expected_ata = anchor_spl::associated_token::get_associated_token_address(
87                  &amounts[i].user, // The user's pubkey from the amounts array
88                  &ctx.accounts.usdc_mint.key()
89              );
90
91              require!(
92                  recipient_token_account.key() == expected_ata,
93                  ErrorCode::InvalidTokenAccount
94              );
95
96
97              let transfer_to = if recipient_token_account.data_is_empty() {
98                  // If ATA is not initialized, transfer to fallback address
99                  ctx.accounts.pacifica_fallback.to_account_info()
100             } else {
101                 // Otherwise transfer to the recipient's ATA
102                 recipient_token_account.clone()
103             };
104
105
106             anchor_spl::token::transfer(
107                 CpiContext::new_with_signer(
108                     ctx.accounts.token_program.to_account_info(),
109                     anchor_spl::token::Transfer {
110                         from: ctx.accounts.pacifica_vault.to_account_info(),
111                         to: transfer_to.clone(),
112                         authority: ctx.accounts.central_state.to_account_info(),
113                     },
114                     signer
115                 ),
116                 amounts[i].amount,
117             )?;
118
119
120             emit!(WithdrawEvent {
```

```
121             user: transfer_to.key(),
122             amount: amounts[i].amount,
123             withdraw_id: amounts[i].withdraw_id,
124             timestamp: Clock::get()?.unix_timestamp,
125         });
126     }
127
128
129     emit_cpi!(BatchCompletedEvent {
130         nonce: ctx.accounts.central_state.withdraw_nonce,
131     });
132
133
134     ctx.accounts.central_state.withdraw_nonce += 1;
135
136
137     Ok(())
138 }
```

Listing 2.1: lib.rs

Impact A single invalid withdrawal entry can prevent the processing of valid user withdrawals, resulting in a denial-of-service scenario.

Suggestion Pre-validate inputs before execution or isolate failures to ensure one invalid entry does not revert the entire batch.

Feedback from the project Based on the above, if a user's receiving account is absent, our contract will redirect the funds to a collector account, ensuring that it does not prevent other users from making withdrawals.

2.1.2 Lack of minimum value check in function `deposit()`

Severity Low

Status Confirmed

Introduced by `Version 1`

Description According to the protocol documentation, the minimum deposit amount for users is `$10`. However, the `deposit()` function does not enforce this check. As a result, users can submit deposits below the documented threshold, creating a mismatch between the intended protocol rules and the on-chain logic. This may lead to unexpected behavior or confusion among users relying on the documented minimum value.

```
35 pub fn deposit(ctx: Context<Deposit>, amount: u64) -> Result<()> {
36     require!(!ctx.accounts.central_state.paused, ErrorCode::CentralStatePaused);
37
38
39     // Transfer USDC from user's token account to Pacifica vault
40     anchor_spl::token::transfer(
41         CpiContext::new(
42             ctx.accounts.token_program.to_account_info(),
43             anchor_spl::token::Transfer {
```

```
44                 from: ctx.accounts.depositor_usdc_account.to_account_info(),
45                 to: ctx.accounts.pacifica_vault.to_account_info(),
46                 authority: ctx.accounts.depositor.to_account_info(),
47             },
48         ),
49         amount,
50     )?;
51
52
53     let current_deposit_nonce = ctx.accounts.central_state.deposit_nonce;
54
55
56     emit_cpi!(DepositEvent {
57         user: ctx.accounts.depositor.key(),
58         amount: amount,
59         timestamp: Clock::get()?.unix_timestamp,
60         deposit_nonce: current_deposit_nonce,
61     });
62
63
64     ctx.accounts.central_state.deposit_nonce += 1;
65
66
67     Ok(())
68 }
```

Listing 2.2: lib.rs

Impact The code allows deposits below the stated minimum value (i.e., $10), creating inconsistency between protocol documentation and actual behavior.

Suggestion Add a check to ensure the input `amount` is greater than or equal to the minimum value allowed by the protocol.

2.2 Recommendations

2.2.1 Redundant checks and unused error codes

Status Confirmed

Introduced by `Version 1`

Description The protocol includes internal checks for the paused state in functions such as `deposit()` and `batch_withdraw()`, even though this condition is already enforced at the context level using `Anchor` constraints. These duplicate validations are unnecessary and could be removed to simplify the logic.

Additionally, some error codes like `MintLimitExceeded` and `InvalidOwner` are defined in the `ErrorCode` enumeration but are never used throughout the program, which are also redundant.

```
241 pub struct Deposit<'info> {
242     #[account(mut)]
243     pub depositor: Signer<'info>,
```

```
244         #[account(
245             mut,
246             associated_token::mint = usdc_mint,
247             associated_token::authority = depositor
248         )]
249         pub depositor_usdc_account: Account<'info, TokenAccount>,
250         #[account(
251             mut,
252             constraint = !central_state.paused @ ErrorCode::CentralStatePaused,
253         )]
254         pub central_state: Account<'info, CentralState>,
255         #[account(
256             mut,
257             associated_token::mint = usdc_mint,
258             associated_token::authority = central_state
259         )]
260         pub pacifica_vault: Account<'info, TokenAccount>,
261         pub token_program: Program<'info, Token>,
262         pub associated_token_program: Program<'info, AssociatedToken>,
263         #[account(
264             constraint = usdc_mint.key() == USDC_MINT_ADDRESS @ ErrorCode::InvalidMint
265         )]
266         pub usdc_mint: Account<'info, Mint>,
267         pub system_program: Program<'info, System>,
268     }
```

Listing 2.3: lib.rs

```
35  pub fn deposit(ctx: Context<Deposit>, amount: u64) -> Result<()> {
36      require!(!ctx.accounts.central_state.paused, ErrorCode::CentralStatePaused);
37
38
39      // Transfer USDC from user's token account to Pacifica vault
40      anchor_spl::token::transfer(
41          CpiContext::new(
42              ctx.accounts.token_program.to_account_info(),
43              anchor_spl::token::Transfer {
44                  from: ctx.accounts.depositor_usdc_account.to_account_info(),
45                  to: ctx.accounts.pacifica_vault.to_account_info(),
46                  authority: ctx.accounts.depositor.to_account_info(),
47              },
48          ),
49          amount,
50      )?;
51
52
53      let current_deposit_nonce = ctx.accounts.central_state.deposit_nonce;
54
55
56      emit_cpi!(DepositEvent {
57          user: ctx.accounts.depositor.key(),
58          amount: amount,
59          timestamp: Clock::get()?.unix_timestamp,
```

```
60          deposit_nonce: current_deposit_nonce,
61      });
62
63
64      ctx.accounts.central_state.deposit_nonce += 1;
65
66
67      Ok(())
68  }
```

Listing 2.4: lib.rs

```
65  pub fn batch_withdraw<'a, 'b, 'c, 'info>(
66      ctx: Context<'a, 'b, 'c, 'info, BatchWithdraw<'info>>,
67      amounts: Vec<UserWithdraw>,
68      nonce: u64
69  ) -> Result<()> {
70      require!(!ctx.accounts.central_state.paused, ErrorCode::CentralStatePaused);
71      require!(amounts.len() == ctx.remaining_accounts.len(), ErrorCode::MismatchingLengths);
72      require!(nonce == ctx.accounts.central_state.withdraw_nonce, ErrorCode::InvalidNonce);
73
74
75      let signer_seeds = &[
76          b"central_state".as_ref(),
77          &[ctx.bumps.central_state]
78      ];
79      let signer = &[&signer_seeds[..]];
80
81
82      for (i, recipient_token_account) in ctx.remaining_accounts.iter().enumerate() {
83
84
85          // Validate that the token account is the correct ATA for the user
86          let expected_ata = anchor_spl::associated_token::get_associated_token_address(
87              &amounts[i].user, // The user's pubkey from the amounts array
88              &ctx.accounts.usdc_mint.key()
89          );
90
91          require!(
92              recipient_token_account.key() == expected_ata,
93              ErrorCode::InvalidTokenAccount
94          );
95
96
97          let transfer_to = if recipient_token_account.data_is_empty() {
98              // If ATA is not initialized, transfer to fallback address
99              ctx.accounts.pacifica_fallback.to_account_info()
100         } else {
101             // Otherwise transfer to the recipient's ATA
102             recipient_token_account.clone()
103         };
104
105
```

```
106             anchor_spl::token::transfer(
107                 CpiContext::new_with_signer(
108                     ctx.accounts.token_program.to_account_info(),
109                     anchor_spl::token::Transfer {
110                         from: ctx.accounts.pacifica_vault.to_account_info(),
111                         to: transfer_to.clone(),
112                         authority: ctx.accounts.central_state.to_account_info(),
113                     },
114                     signer
115                 ),
116                 amounts[i].amount,
117             )?;
118
119
120             emit!(WithdrawEvent {
121                 user: transfer_to.key(),
122                 amount: amounts[i].amount,
123                 withdraw_id: amounts[i].withdraw_id,
124                 timestamp: Clock::get()?.unix_timestamp,
125             });
126         }
127
128
129         emit_cpi!(BatchCompletedEvent {
130             nonce: ctx.accounts.central_state.withdraw_nonce,
131         });
132
133
134         ctx.accounts.central_state.withdraw_nonce += 1;
135
136
137         Ok(())
138     }
```

Listing 2.5: lib.rs

```
272 pub struct BatchWithdraw<'info> {
273     #[account(
274         mut,
275         constraint = withdraw_authority.key() == central_state.withdraw_authority @ ErrorCode::
                InvalidWithdrawAuthority
276     )]
277     pub withdraw_authority: Signer<'info>,
278     #[account(
279         mut,
280         constraint = !central_state.paused @ ErrorCode::CentralStatePaused,
281         seeds = [b"central_state"],
282         bump
283     )]
284     pub central_state: Account<'info, CentralState>,
285     #[account(
286         mut,
287         associated_token::mint = usdc_mint,
```

```
288             associated_token::authority = central_state
289         )]
290         pub pacifica_vault: Account<'info, TokenAccount>,
291         #[account(
292             mut,
293             seeds = [b"pacifica_fallback"],
294             bump
295         )]
296         pub pacifica_fallback: Account<'info, TokenAccount>,
297         pub token_program: Program<'info, Token>,
298         pub associated_token_program: Program<'info, AssociatedToken>,
299         #[account(
300             constraint = usdc_mint.key() == USDC_MINT_ADDRESS @ ErrorCode::InvalidMint)]
301         pub usdc_mint: Account<'info, Mint>,
302         pub system_program: Program<'info, System>,
303         // remaining accounts are the withdraw recipients
304     }
```

Listing 2.6: lib.rs

```
272     #[error_code]
273     pub enum ErrorCode {
274         #[msg("Invalid mint")]
275         InvalidMint,
276         #[msg("Central state is paused")]
277         CentralStatePaused,
278         #[msg("Invalid withdraw authority")]
279         InvalidWithdrawAuthority,
280         #[msg("Mismatching lengths")]
281         MismatchingLengths,
282         #[msg("Invalid token account")]
283         InvalidTokenAccount,
284         #[msg("Invalid owner")]
285         InvalidOwner,
286         #[msg("Invalid nonce")]
287         InvalidNonce,
288         #[msg("Invalid program authority")]
289         InvalidProgramAuthority,
290         #[msg("Invalid pause authority")]
291         InvalidPauseAuthority,
292         #[msg("Mint limit exceeded")]
293         MintLimitExceeded,
294     }
```

Listing 2.7: lib.rs

Suggestion Remove redundant codes.

2.2.2 Lack of explicit error messages for access control violations

Status Confirmed

Introduced by Version 1

Description Several instruction context structures, including `Unpause`, `SetWithdrawAuthority`, and `WithdrawFromFallback`, rely on `has_one` constraints to ensure that only specific authorities can execute the corresponding instructions. However, when a `signer` fails to meet these constraints, the transaction is rejected without returning a clear error message.

```
319  pub struct Unpause<'info> {
320      #[account(mut)]
321      pub program_authority: Signer<'info>,
322      #[account(
323          mut,
324          has_one = program_authority
325      )]
326      pub central_state: Account<'info, CentralState>,
327  }
```

Listing 2.8: lib.rs

```
330  pub struct SetWithdrawAuthority<'info> {
331      #[account(mut)]
332      pub program_authority: Signer<'info>,
333      #[account(
334          mut,
335          has_one = program_authority
336      )]
337      pub central_state: Account<'info, CentralState>,
338  }
```

Listing 2.9: lib.rs

```
352  pub struct WithdrawFromFallback<'info> {
353      pub withdraw_authority: Signer<'info>,
354      #[account(
355          has_one = withdraw_authority,
356          seeds = [b"central_state"],
357          bump
358      )]
359      pub central_state: Account<'info, CentralState>,
360      #[account(
361          mut,
362          seeds = [b"pacifica-fallback"],
363          bump
364      )]
365      pub pacifica_fallback: Account<'info, TokenAccount>,
366      /// CHECK: Recipient account
367      pub recipient: AccountInfo<'info>,
368      #[account(
369          mut,
370          associated_token::mint = usdc_mint,
371          associated_token::authority = recipient
372      )]
373      pub recipient_token_account: Account<'info, TokenAccount>,
374      #[account(
375          constraint = usdc_mint.key() == USDC_MINT_ADDRESS @ ErrorCode::InvalidMint
```

```
376         )]
377         pub usdc_mint: Account<'info, Mint>,
378         pub associated_token_program: Program<'info, AssociatedToken>,
379         pub token_program: Program<'info, Token>,
380         pub system_program: Program<'info, System>,
381     }
```

Listing 2.10: lib.rs

Suggestion Add explicit error messages to improve clarity and protocol usability.

2.2.3 Lack of nonce check in function `deposit()`

Status Confirmed

Introduced by `Version 1`

Description The protocol maintains separate counters (`deposit_nonce` and `withdraw_nonce`) in `central_state` to track deposit and withdrawal operations. In the `batch_withdraw()` function, the `withdraw_nonce` is compared with a `nonce` provided by the signer to ensure alignment between user requests and the program's internal state.

However, the `deposit()` function does not include a similar check for `deposit_nonce`. While this does not directly impact the correctness of the deposit logic, it introduces an potential inconsistency in how deposit and withdrawal operations are handled.

```
65  pub fn batch_withdraw<'a, 'b, 'c, 'info>(
66      ctx: Context<'a, 'b, 'c, 'info, BatchWithdraw<'info>>,
67      amounts: Vec<UserWithdraw>,
68      nonce: u64
69  ) -> Result<()> {
70      require!(!ctx.accounts.central_state.paused, ErrorCode::CentralStatePaused);
71      require!(amounts.len() == ctx.remaining_accounts.len(), ErrorCode::MismatchingLengths);
72      require!(nonce == ctx.accounts.central_state.withdraw_nonce, ErrorCode::InvalidNonce);
73
74
75      let signer_seeds = &[
76          b"central_state".as_ref(),
77          &[ctx.bumps.central_state]
78      ];
79      let signer = &[&signer_seeds[..]];
80
81
82      for (i, recipient_token_account) in ctx.remaining_accounts.iter().enumerate() {
83
84
85          // Validate that the token account is the correct ATA for the user
86          let expected_ata = anchor_spl::associated_token::get_associated_token_address(
87              &amounts[i].user, // The user's pubkey from the amounts array
88              &ctx.accounts.usdc_mint.key()
89          );
90
91          require!(
92              recipient_token_account.key() == expected_ata,
```

```
93              ErrorCode::InvalidTokenAccount
94          );
95
96
97          let transfer_to = if recipient_token_account.data_is_empty() {
98              // If ATA is not initialized, transfer to fallback address
99              ctx.accounts.pacifica_fallback.to_account_info()
100         } else {
101             // Otherwise transfer to the recipient's ATA
102             recipient_token_account.clone()
103         };
104
105
106         anchor_spl::token::transfer(
107             CpiContext::new_with_signer(
108                 ctx.accounts.token_program.to_account_info(),
109                 anchor_spl::token::Transfer {
110                     from: ctx.accounts.pacifica_vault.to_account_info(),
111                     to: transfer_to.clone(),
112                     authority: ctx.accounts.central_state.to_account_info(),
113                 },
114                 signer
115             ),
116             amounts[i].amount,
117         )?;
118
119
120         emit!(WithdrawEvent {
121             user: transfer_to.key(),
122             amount: amounts[i].amount,
123             withdraw_id: amounts[i].withdraw_id,
124             timestamp: Clock::get()?.unix_timestamp,
125         });
126     }
127
128
129     emit_cpi!(BatchCompletedEvent {
130         nonce: ctx.accounts.central_state.withdraw_nonce,
131     });
132
133
134     ctx.accounts.central_state.withdraw_nonce += 1;
135
136
137     Ok(())
138 }
```

Listing 2.11: lib.rs

```
35  pub fn deposit(ctx: Context<Deposit>, amount: u64) -> Result<()> {
36      require!(!ctx.accounts.central_state.paused, ErrorCode::CentralStatePaused);
37
38
```

```
39      // Transfer USDC from user's token account to Pacifica vault
40      anchor_spl::token::transfer(
41          CpiContext::new(
42              ctx.accounts.token_program.to_account_info(),
43              anchor_spl::token::Transfer {
44                  from: ctx.accounts.depositor_usdc_account.to_account_info(),
45                  to: ctx.accounts.pacifica_vault.to_account_info(),
46                  authority: ctx.accounts.depositor.to_account_info(),
47              },
48          ),
49          amount,
50      )?;
51
52
53      let current_deposit_nonce = ctx.accounts.central_state.deposit_nonce;
54
55
56      emit_cpi!(DepositEvent {
57          user: ctx.accounts.depositor.key(),
58          amount: amount,
59          timestamp: Clock::get()?.unix_timestamp,
60          deposit_nonce: current_deposit_nonce,
61      });
62
63
64      ctx.accounts.central_state.deposit_nonce += 1;
65
66
67      Ok(())
68  }
```

Listing 2.12: lib.rs

Suggestion Add a nonce parameter to function `deposit()` and check it against `deposit_nonce` to align with the design pattern used in withdrawals.

2.2.4 Lack of state check in functions `pause()` and `unpause()`

Status Confirmed

Introduced by `Version 1`

Description The privileged functions `pause()` and `unpause()` are designed to toggle the protocol's `deposit` and `withdrawal` functionality by switching its paused state. However, the current implementation does not verify the program's existing state before performing the action. This means that the function `pause()` can be invoked even when the program is already paused, and the function `unpause()` can be invoked even when it's already unpaused. Although this does not directly cause a functional error, it may lead to unnecessary transactions or obscure the program's operational state transitions, which could hinder debugging or monitoring.

```
131  pub fn pause(ctx: Context<Pause>) -> Result<()> {
132      let old_state = ctx.accounts.central_state.paused;
133      ctx.accounts.central_state.paused = true;
```

```
134
135
136         emit!(PauseEnabledEvent {
137             old_state: old_state,
138             new_state: true,
139         });
140
141
142         Ok(())
143     }
144
145
146     pub fn unpause(ctx: Context<Unpause>) -> Result<()> {
147         ctx.accounts.central_state.paused = false;
148
149
150         emit!(PauseDisabledEvent {
151             old_state: true,
152             new_state: false,
153         });
154
155
156         Ok(())
157     }
```

Listing 2.13: lib.rs

Suggestion Add checks to ensure functions `pause()` and `unpause()` are only executed when the program is in the opposite state.

2.3 Notes

2.3.1 Potential centralization risk

Introduced by `Version 1`

Description In the current implementation, privileged roles (e.g., parameter setting, pause/unpause and grant roles) are set to govern and regulate the system-wide operation . Additionally, accounts with `withdraw_authority` permission can withdraw user assets from the vault unlimitedly. If the private keys of them are lost or maliciously exploited, it could potentially lead to losses for users.

2.3.2 Potential risk of front-running in contract initialization

Introduced by `Version 1`

Description The `initialize()` function sets critical roles (such as `program_authority` and `withdraw_authority`) in the `central_state` account, which is derived from fixed `seeds` and can only be created once. Since the function lacks access control, there's a risk that a malicious user could front-run the intended deployment and initialize the protocol with unauthorized

values. In such cases, the program would have to be redeployed to recover control. To avoid this, ensure the contract is initialized immediately after deployment.


BOOST WEB3 THROUGH NEXT-GENERATION SECURITY & USABILITY INNOVATIONS