Security Assessment Report

Keel SVM ALM Controller

October 2, 2025

Summary

The Sec3 team was engaged to conduct a thorough security analysis of the Keel SVM ALM Controller.

The artifact of the audit was the source code of the following programs, excluding tests, in a private repository.

The initial audit focused on the following versions and revealed 30 issues or questions.

#	Task	Type	Commit
P1	Commit ddf437f	Solana	`ddf437f` (commit date: Jul 15, 2025)
P2	Commit c3836e1	Solana	`c3836e1` (commit date: Aug 08, 2025)

The post-audit was conducted on version `cf022cb` (commit date: Sep 23, 2025), which concludes this audit.

This report provides a detailed description of the findings and their respective resolutions.

Table of Contents

Result Overview

Issue	Impact	Status
COMMIT DDF437F		
[P1-L-01] Missing program id validation in integration initialization	Low	Resolved
[P1-L-02] Missing mint account validation against configuration	Low	Resolved
[P1-L-03] Possible outflow limit bypass	Low	Resolved
[P1-L-04] Multiple push instructions for a single send leads to fund theft	Low	Resolved
[P1-L-05] Possible integer overflow	Low	Resolved
[P1-L-06] Missing checks for the `PullFeedAccountData` account	Low	Resolved
[P1-I-01] Redundant checks on mint	Info	Resolved
[P1-I-02] Typos	Info	Resolved
[P1-I-03] Wrong accounts checked in `ManagePermissionAccounts::checked_from_accounts`	Info	Resolved
[P1-I-04] Ensure `input_mint != output_mint`	Info	Resolved
[P1-I-05] Insufficient ownership check for `lp_token_account`	Info	Resolved
[P1-I-06] Inconsistent variable naming in `PullSplTokenSwapAccounts`	Info	Resolved
[P1-I-07] Arbitrage opportunities due to slippage-based repayments	Info	Acknowledged
[P1-I-08] Potential DoS due to rounding issue in `refresh_rate_limit`	Info	Resolved
[P1-I-09] Unnecessary `load_and_check_mut`	Info	Resolved
[P1-I-10] Unnecessary `drop()`	Info	Resolved
[P1-I-11] Missing signer check for `message_sent_event_data`	Info	Resolved
[P1-I-12] Arbitrage enabled by improper single token withdraw parameters	Info	Resolved
[P1-I-13] Unused accounts	Info	Resolved
[P1-I-14] Increment may be truncated to a small value in `refresh_rate_limit`	Info	Resolved
COMMIT C3836E1		
[P2-L-01] Missing mint extension validation in integration initialization	Low	Resolved
[P2-L-02] Missing Token-2022 `Mint` and `TokenAccount` type checks	Low	Resolved
[P2-L-03] Potential DoS caused by `PermanentDelegate` extension	Low	Resolved
[P2-L-04] Inaccurate inflow accounting if `TransferFeeConfig` is enabled	Low	Resolved
[P2-I-01] `AtomicSwap` should reject mints with the `MintCloseAuthority` extension	Info	Acknowledged
[P2-I-02] `MemoTransfer` extension is not a mint extension	Info	Resolved
[P2-I-03] Missing `MemoTransfer` extension support in `transfer_tokens`	Info	Acknowledged
[P2-I-04] Missing mint `Pausable` extension validation	Info	Resolved
[P2-I-05] Check `payer_account_a` mint in `atomic_swap_repay`	Info	Acknowledged
[P2-Q-01] Question on the `ConfidentialTransferMint` support	Question	Resolved

Findings in Detail

COMMIT DDF437F

[P1-L-01] Missing program id validation in integration initialization

Identified in commit `ddf437f`.

The integration refers to the configuration to interact with other programs.

During the initialization of the `spl_token_swap`, `cctp_bridge`, and `lz_bridge` integrations, the provided program accounts are not verified to be the expected Nova PSM, CCTP, or LayerZero Bridge programs.

```
/* program/src/integrations/spl_token_swap/initialize.rs */
133 | program: Pubkey::from(*inner_ctx.swap_program.key()),

/* program/src/integrations/cctp_bridge/initialize.rs */
085 | cctp_token_messenger_minter: Pubkey::from(*inner_ctx.cctp_token_messenger_minter.key()),
086 | cctp_message_transmitter: Pubkey::from(*inner_ctx.cctp_message_transmitter.key()),

/* program/src/integrations/lz_bridge/initialize.rs */
092 | program: Pubkey::from(*inner_ctx.lz_program.key()),
```

It is recommended to add whitelist verification to ensure that these programs are expected ones.

Resolution

This issue has been fixed by `5f65360`.

COMMIT DDF437F

[P1-L-02] Missing mint account validation against configuration

Identified in commit `ddf437f`.

In `process_push_spl_token_swap` and `process_pull_spl_token_swap`, checks for `mint_a` and `mint_b` are missing.

The current code only requires `mint_a` and `mint_b` to match `reserve_a.mint` and `reserve_b.mint`. It does not compare them with the mints recorded in the integration's configuration. Since there is no validation for `reserve_a` and `reserve_b` either, they can be any reserves managed by the controller. This means `mint_a` and `mint_b` could be the mints for any reserve, not necessarily the two mints intended for the current swap integration. This also allows `vault_a` and `vault_b` to be vaults for any mint.

```
/* program/src/integrations/spl_token_swap/push.rs */
126 | pub fn process_push_spl_token_swap(
134 | ) -> Result<(), ProgramError> {
          // @audit: missing check to ensure `inner_ctx.mint_a == integration.config.mint_a && inner_ctx.mint_b ==
          ↪  integration.config.mint_b`
162 |     if inner_ctx.vault_a.key().ne(&reserve_a.vault) {
163 |         msg! {"mint_a: mismatch with reserve"};
164 |         return Err(ProgramError::InvalidAccountData);
165 |     }
166 |     if inner_ctx.vault_b.key().ne(&reserve_b.vault) {
167 |         msg! {"vault_b: mismatch with reserve"};
168 |         return Err(ProgramError::InvalidAccountOwner);
169 |     }
170 |     if inner_ctx.mint_a.key().ne(&reserve_a.mint) {
171 |         msg! {"mint_a: mismatch with reserve"};
172 |         return Err(ProgramError::InvalidAccountData);
173 |     }
174 |     if inner_ctx.mint_b.key().ne(&reserve_b.mint) {
175 |         msg! {"mint_b: mismatch with reserve"};
176 |         return Err(ProgramError::InvalidAccountData);
177 |     }
465 | }

/* program/src/integrations/spl_token_swap/pull.rs */
126 | pub fn process_pull_spl_token_swap(
134 | ) -> Result<(), ProgramError> {
          // @audit: The same problem exists here.
162 |     if inner_ctx.vault_a.key().ne(&reserve_a.vault) {
163 |         msg! {"mint_a: mismatch with reserve"};
164 |         return Err(ProgramError::InvalidAccountData);
165 |     }
166 |     if inner_ctx.vault_b.key().ne(&reserve_b.vault) {
167 |         msg! {"vault_b: mismatch with reserve"};
168 |         return Err(ProgramError::InvalidAccountData);
169 |     }
170 |     if inner_ctx.mint_a.key().ne(&reserve_a.mint) {
171 |         msg! {"mint_a: mismatch with reserve"};
172 |         return Err(ProgramError::InvalidAccountData);
```

```
173 |     }
174 |     if inner_ctx.mint_b.key().ne(&reserve_b.mint) {
175 |         msg! {"mint_b: mismatch with reserve"};
176 |         return Err(ProgramError::InvalidAccountData);
177 |     }
467 | }
```

We can also see that `swap_token_a` and `swap_token_b` have proper checks requiring them to match the accounts stored in the swap state. However, the mints of these token accounts are not linked to `mint_a` and `mint_b`. This makes it possible for `swap_token_a` and `swap_token_b` to be correct while `mint_a` and `mint_b` are wrong. For example, `mint_a` and `mint_b` could be swapped, resulting in a situation where `swap_token_a.mint == mint_b` and `swap_token_b.mint == mint_a`. This could still pass the checks in `deposit_single_token_type_exact_amount_in_cpi` / `withdraw_single_token_type_exact_amount_out_cpi`.

```
/* program/src/integrations/spl_token_swap/push.rs */
126 | pub fn process_push_spl_token_swap(
134 | ) -> Result<(), ProgramError> {
181 |     let swap_data = inner_ctx.swap.try_borrow_data()?;
182 |     let swap_state = SwapV1Subset::try_from_slice(&swap_data[1..LEN_SWAP_V1_SUBSET + 1]).unwrap();
189 |     if swap_state.token_a.ne(inner_ctx.swap_token_a.key()) {
190 |         msg! {"swap_token_a: does not match swap state"};
191 |         return Err(ProgramError::InvalidAccountData);
192 |     }
193 |     if swap_state.token_b.ne(inner_ctx.swap_token_b.key()) {
194 |         msg! {"swap_token_b: does not match swap state"};
195 |         return Err(ProgramError::InvalidAccountData);
196 |     }
465 | }

/* program/src/integrations/spl_token_swap/pull.rs */
126 | pub fn process_pull_spl_token_swap(
134 | ) -> Result<(), ProgramError> {
181 |     let swap_data = inner_ctx.swap.try_borrow_data()?;
182 |     let swap_state = SwapV1Subset::try_from_slice(&swap_data[1..LEN_SWAP_V1_SUBSET + 1]).unwrap();
189 |     if swap_state.token_a.ne(inner_ctx.swap_token_a.key()) {
190 |         msg! {"swap_token_a: does not match swap state"};
191 |         return Err(ProgramError::InvalidAccountData);
192 |     }
193 |     if swap_state.token_b.ne(inner_ctx.swap_token_b.key()) {
194 |         msg! {"swap_token_b: does not match swap state"};
195 |         return Err(ProgramError::InvalidAccountData);
196 |     }
467 | }
```

This issue does not currently cause a major problem. This is because these two functions do not use the reserve or vault information to update any state. Since the swap account determines the two types of tokens, it is not possible to use the `deposit_single_token_type_exact_amount_in_cpi` to move funds from other reserves into the swap pool.

The only part that is affected is `emit_event`, which will contain the wrong mint. This could impact off

chain programs that rely on these events.

```
/* program/src/integrations/spl_token_swap/push.rs */
126 | pub fn process_push_spl_token_swap(
134 | ) -> Result<(), ProgramError> {
410 |     // Emit the accounting event
411 |     if step_2_balance_a != post_deposit_balance_a {
412 |         controller.emit_event(
413 |             outer_ctx.controller_authority,
414 |             outer_ctx.controller.key(),
415 |             SvmAlmControllerEvent::AccountingEvent(AccountingEvent {
416 |                 controller: *outer_ctx.controller.key(),
417 |                 integration: *outer_ctx.integration.key(),
                    // @audit: mint_a could be an incorrect mint account.
418 |                 mint: *inner_ctx.mint_a.key(),
419 |                 action: AccountingAction::Deposit,
420 |                 before: step_2_balance_a,
421 |                 after: post_deposit_balance_a,
422 |             }),
423 |         )?;
424 |     }
465 | }
```

It is recommended to add checks to verify that `mint_a` and `mint_b` match `config.mint_a` and `config.mint_b`. Additionally, the mints for `swap_token_a` and `swap_token_b` should also be required to match them.

Resolution

This issue has been fixed by `8fc0883` and `c873d3a`.

COMMIT DDF437F

[P1-L-03] Possible outflow limit bypass

Identified in commit `ddf437f`.

The `Reserve`'s vault and the `SplTokenSwap` integration's `lp_token_account` are both ATAs with the same `controller_authority`. This means if a `Reserve` is created for the same mint as a `SplTokenSwap` integration's `lp_mint` under the same controller, their funds will be held in the same token account. This could lead to incorrect inflow calculations and may cause the `Reserve`'s outflow limit to fail.

```
/* program/src/integrations/spl_token_swap/initialize.rs */
078 | pub fn process_initialize_spl_token_swap(
081 | ) -> Result<(IntegrationConfig, IntegrationState), ProgramError> {
120 |     CreateIdempotent {
121 |         funding_account: outer_ctx.payer,
              // N: The lp_token_account is an ATA, and its authority is the controller_authority.
122 |         account: inner_ctx.lp_token_account,
123 |         wallet: outer_ctx.controller_authority,
124 |         mint: inner_ctx.lp_mint,
125 |         system_program: outer_ctx.system_program,
126 |         token_program: inner_ctx.lp_mint_token_program,
127 |     }
128 |     .invoke()
129 |     .unwrap();
130 |
131 |     // Create the Config
132 |     let config = IntegrationConfig::SplTokenSwap(SplTokenSwapConfig {
137 |         lp_mint: Pubkey::from(*inner_ctx.lp_mint.key()),
              // @audit: The lp_token_account is stored in the IntegrationConfig
              //         and will be used later to receive and store LP tokens.
138 |         lp_token_account: Pubkey::from(*inner_ctx.lp_token_account.key()),
140 |     });
166 | }

/* program/src/processor/initialize_reserve.rs */
030 | pub fn process_initialize_reserve(
034 | ) -> ProgramResult {
063 |     CreateIdempotent {
064 |         funding_account: ctx.payer,
              // N: The reserve's vault is an ATA, and its authority is also the controller_authority.
065 |         account: ctx.vault,
066 |         wallet: ctx.controller_authority,
067 |         mint: ctx.mint,
068 |         system_program: ctx.system_program,
069 |         token_program: ctx.token_program,
070 |     }
071 |     .invoke()
072 |     .unwrap();
111 | }
```

First, `process_pull_spl_token_swap` contains logic to call the swap program to withdraw `token_a` or `token_b` using LP tokens. However, it does not consider that the `lp_mint` might also have a corresponding `Res`

`erve`. The amount of LP tokens spent is not counted as part of the outflow for that `Reserve`, which could bypass its flow control.

```
/* program/src/integrations/spl_token_swap/pull.rs */
126 | pub fn process_pull_spl_token_swap(
134 | ) -> Result<(), ProgramError> {
          // @audit: This calls the swap program to withdraw token_a or
          //         token_b by burning LP tokens.
336 |     if amount_a > 0 {
337 |         withdraw_single_token_type_exact_amount_out_cpi(
357 |         )?;
358 |     }
359 |     if amount_b > 0 {
360 |         withdraw_single_token_type_exact_amount_out_cpi(
380 |         )?;
381 |     }
          // @audit: However, changes in the LP token balance only affect the
          //         integration's flow control. If this lp_mint has a corresponding
          //         Reserve, its state will not be updated.
456 |     integration.update_rate_limit_for_inflow(clock, delta_lp as u64)?;
457 |
458 |     // Update the reserves for the flows
459 |     if amount_a > 0 {
460 |         reserve_a.update_for_inflow(clock, amount_a)?;
461 |     }
462 |     if amount_b > 0 {
463 |         reserve_b.update_for_inflow(clock, amount_b)?;
464 |     }
465 |
466 |     Ok(())
467 | }
```

There are two possible ways to fix this.

One solution is to separate the accounts completely by changing the `SplTokenSwap` integration's `lp_token_account` to an ATA that does not belong to the `controller_authority`.

Another approach is to bring the `lp_token_account` under the `Reserve`'s management. This would require the `lp_token_account` to be a `Reserve`'s vault and for the `Reserve` to be updated after each operation.

Resolution

This issue has been fixed by `6d6131e`.

COMMIT DDF437F

[P1-L-04] Multiple push instructions for a single send leads to fund theft

Identified in commit `ddf437f`.

In `process_push_lz_bridge`, the program transfers funds to the `authority_token_account`, which is an ATA for the `permission.authority`. It also checks that the same transaction includes a `send` CPI call to the `lz_program`. This check ensures that the accounts, target chain, destination address, and amount all match the requirements in `integration.config` and the original request. The goal is to make sure that the funds transferred to the `authority_token_account` are then correctly sent to the `lz_program` and then send to the target account on the destination chain.

The problem is that the program does not check if there is a `process_push_lz_bridge` instruction within the same transaction. This means it is possible for a single `lz_program` send instruction to correspond to multiple `process_push_lz_bridge` calls.

In this situation, the program would transfer funds to the `authority_token_account` multiple times, but it would only need to perform the `send` operation once. The remaining funds could be kept in the authority's account, leading to the theft of funds from the vault.

```
/* program/src/integrations/lz_bridge/push.rs */
058 | pub fn verify_send_ix_in_tx(
063 | ) -> ProgramResult {
          // @audit: The checks below confirm a 'send' instruction exists in the
          //         transaction, but they do not prevent multiple 'process_push_lz_bridge'
          //         instructions from being included.
064 |     // Get number of instructions in current transaction.
065 |     let sysvar_data = accounts.sysvar_instruction.try_borrow_data()?;
066 |     if sysvar_data.len() < 2 {
067 |         return Err(SvmAlmControllerErrors::InvalidInstructions.into());
068 |     }
069 |     let ix_len = u16::from_le_bytes([sysvar_data[0], sysvar_data[1]]);
070 |
071 |     let instructions = Instructions::try_from(accounts.sysvar_instruction)?;
072 |
073 |     // Check that current ix is before the last ix.
074 |     let curr_ix = instructions.load_current_index();
075 |     if curr_ix >= ix_len - 1 {
076 |         return Err(SvmAlmControllerErrors::UnauthorizedAction.into());
077 |     }
078 |
079 |     // Load last instruction in transaction and check that its for OFT program.
080 |     let last_ix = instructions.load_instruction_at((ix_len - 1).into())?;
081 |     if last_ix.get_program_id().ne(&config.program) {
082 |         return Err(SvmAlmControllerErrors::InvalidInstructions.into());
083 |     }
084 |
085 |     // Deserializes and checks that ix discriminator matches known send_ix discriminator.
```

```
086 |     let send_args = OftSendParams::deserialize(last_ix.get_instruction_data())?;
087 |
088 |     let signer = last_ix.get_account_meta_at(0)?.key;
089 |     let peer_config = last_ix.get_account_meta_at(1)?.key;
090 |     let oft_store = last_ix.get_account_meta_at(2)?.key;
091 |     let token_source = last_ix.get_account_meta_at(3)?.key;
092 |     let token_escrow = last_ix.get_account_meta_at(4)?.key;
093 |     let token_mint = last_ix.get_account_meta_at(5)?.key;
094 |     let token_program = last_ix.get_account_meta_at(6)?.key;
095 |
096 |     // Check that accounts for send_ix matches known accounts.
097 |     if signer.ne(authority)
098 |         || peer_config.ne(&config.peer_config)
099 |         || oft_store.ne(&config.oft_store)
100 |         || token_source.ne(accounts.authority_token_account.key())
101 |         || token_escrow.ne(&config.token_escrow)
102 |         || token_mint.ne(accounts.mint.key())
103 |         || token_program.ne(accounts.token_program.key())
104 |     {
105 |         return Err(SvmAlmControllerErrors::InvalidInstructions.into());
106 |     }
107 |
108 |     // Check that ix args for send_ix matches known values.
109 |     if send_args.amount_ld != amount
110 |         || send_args.to != config.destination_address
111 |         || send_args.dst_eid != config.destination_eid
112 |     {
113 |         return Err(SvmAlmControllerErrors::InvalidInstructions.into());
114 |     }
115 |
116 |     Ok(())
117 | }
118 |
119 | pub fn process_push_lz_bridge(
126 | ) -> Result<(), ProgramError> {
157 |     verify_send_ix_in_tx(outer_ctx.authority.key(), &inner_ctx, &config, amount)?;
179 |     CreateIdempotent {
180 |         funding_account: outer_ctx.authority,
              // @audit: The 'authority_token_account' is a token account controlled by an
              //         external account ('permission.authority'), not a program-derived address.
              //         The program cannot control how these tokens are used after transfer,
              //         it can only require a matching 'send' to the OFT program.
181 |         account: inner_ctx.authority_token_account,
182 |         wallet: outer_ctx.authority,
183 |         mint: inner_ctx.mint,
184 |         system_program: inner_ctx.system_program,
185 |         token_program: inner_ctx.token_program,
186 |     }
187 |     .invoke()?;
188 |
191 |     controller.transfer_tokens(
192 |         outer_ctx.controller,
193 |         outer_ctx.controller_authority,
194 |         inner_ctx.vault,
195 |         inner_ctx.authority_token_account,
196 |         amount,
197 |     )?;
198 |
235 | }
```

Because `process_push_lz_bridge` requires the `can_invoke_external_transfer` permission, regular users cannot launch an attack, so the security risk is relatively low. However, this vulnerability undermines a key security principle of the program. The `can_invoke_external_transfer` permission was designed only to allow transfers to trusted targets configured by `can_manage_integrations`, preventing anyone from intercepting the funds. This vulnerability breaks that security design and makes it possible to steal funds.

Since `process_push_lz_bridge` can also be called via a CPI, simply requiring only one `process_push_lz_bridge` instruction in the `sysvar_instruction` is not enough to fix the issue.

One possible solution is to block CPI calls entirely and check that the transaction does not contain any other `push` instructions. This would ensure a one to one relationship between a `process_push_lz_bridge` instruction and an `lz_program` send instruction. The `token-2022` implementation below can be used as a reference for checking if a program is being called via CPI.

```
https://github.com/solana-program/token-2022/blob/2e799e3/program/src/extension/cpi_guard/mod.rs#L39-L51
039 | /// Determine if we are in CPI
040 | pub fn in_cpi() -> bool {
041 |     #[cfg(target_os = "solana")]
042 |     #[allow(unsafe_code)]
043 |     unsafe {
044 |         use solana_instruction::{syscalls::sol_get_stack_height, TRANSACTION_LEVEL_STACK_HEIGHT};
045 |         sol_get_stack_height() as usize > TRANSACTION_LEVEL_STACK_HEIGHT
046 |     }
047 |     #[cfg(not(target_os = "solana"))]
048 |     {
049 |         false
050 |     }
051 | }
```

Resolution

This issue has been fixed by `dbfc4ac`, `78bbda9` and `09b29bd`.

COMMIT DDF437F

[P1-L-05] Possible integer overflow

Identified in commit `ddf437f`.

Similar to the issue "Possible outflow limit bypass", sharing the same ATA for a Reserve's vault and a `SplTokenSwap` integration's `lp_token_account` introduces another problem. In the logic related to `spl_token_swap`, an integer overflow can occur when calculating `step_1_balance_a` and `step_1_balance_b`.

When the two accounts are the same, another integration connected to the Reserve can transfer LP tokens out of the shared `lp_token_account`. This action does not update the `last_balance_lp` value stored in the integration's state. As a result, the `last_balance_lp` value used in calculations might be larger than the account's real balance, and it could even become larger than the total `lp_mint_supply`.

If `last_balance_lp` is greater than `lp_mint_supply`, the ratio of `last_balance_lp / lp_mint_supply` will be greater than 1. This could make the calculated `step_1_balance_a` larger than the `swap_token_a_balance`, potentially exceeding the maximum limit of a `u64` variable. The `as u64` cast in the calculation will then discard the extra bits, leading to an incorrect value for `step_1_balance_a` and `step_1_balance_b`.

Fortunately, these two values only affect the emission of events. This could lead to incorrect amounts in the emitted `AccountingEvent` or cause some `AccountingEvent` not to be emitted at all.

```
/* program/src/integrations/spl_token_swap/pull.rs */
126 | pub fn process_pull_spl_token_swap(
134 | ) -> Result<(), ProgramError> {
216 |     // Extract the values from the last update
217 |     let (last_balance_a, last_balance_b, last_balance_lp) = match integration.state {
218 |         IntegrationState::SplTokenSwap(state) => (
219 |             state.last_balance_a,
220 |             state.last_balance_b,
221 |             state.last_balance_lp as u128,
222 |         ),
223 |         _ => return Err(ProgramError::InvalidAccountData),
224 |     };
227 |     let lp_mint_supply = lp_mint.supply() as u128;
243 |     if last_balance_lp > 0 {
244 |         step_1_balance_a = (swap_token_a_balance as u128 * last_balance_lp / lp_mint_supply) as u64;
245 |         step_1_balance_b = (swap_token_b_balance as u128 * last_balance_lp / lp_mint_supply) as u64;
246 |     } else { // N: assume lp amount unchanged
247 |         step_1_balance_a = 0u64;
248 |         step_1_balance_b = 0u64;
249 |     }
467 | }

/* program/src/integrations/spl_token_swap/push.rs */
126 | pub fn process_push_spl_token_swap(
134 | ) -> Result<(), ProgramError> {
```

```
216 |     // // Extract the values from the last update
217 |     let (last_balance_a, last_balance_b, last_balance_lp) = match integration.state {
218 |         IntegrationState::SplTokenSwap(state) => (
219 |             state.last_balance_a,
220 |             state.last_balance_b,
221 |             state.last_balance_lp as u128,
222 |         ),
223 |         _ => return Err(ProgramError::InvalidAccountData),
224 |     };
227 |     let lp_mint_supply = lp_mint.supply() as u128;
243 |     if last_balance_lp > 0 {
244 |         step_1_balance_a = (swap_token_a_balance as u128 * last_balance_lp / lp_mint_supply) as u64;
245 |         step_1_balance_b = (swap_token_b_balance as u128 * last_balance_lp / lp_mint_supply) as u64;
246 |     } else {
247 |         step_1_balance_a = 0u64;
248 |         step_1_balance_b = 0u64;
249 |     }
465 | }

/* program/src/integrations/spl_token_swap/sync.rs */
067 | pub fn process_sync_spl_token_swap(
071 | ) -> Result<(), ProgramError> {
079 |     let lp_mint_supply = lp_mint.supply() as u128;
105 |     // Extract the values from the last update
106 |     let (last_balance_a, last_balance_b, last_balance_lp) = match integration.state {
107 |         IntegrationState::SplTokenSwap(state) => (
108 |             state.last_balance_a,
109 |             state.last_balance_b,
110 |             state.last_balance_lp as u128,
111 |         ),
112 |         _ => return Err(ProgramError::InvalidAccountData),
113 |     };
124 |     if last_balance_lp > 0 {
125 |         step_1_balance_a =
126 |             (swap_token_a.amount() as u128 * last_balance_lp / lp_mint_supply) as u64;
127 |         step_1_balance_b =
128 |             (swap_token_b.amount() as u128 * last_balance_lp / lp_mint_supply) as u64;
129 |     } else {
130 |         step_1_balance_a = 0u64;
131 |         step_1_balance_b = 0u64;
132 |     }
234 | }
```

Resolution

This issue has been fixed by `6d6131e`.

COMMIT DDF437F

[P1-L-06] Missing checks for the `PullFeedAccountData` account

Identified in commit `ddf437f`.

The program is missing discriminator or owner checks when using the `PullFeedAccountData` account.

In `process_refresh_oracle`, the `price_feed` account data is deserialized starting from the 8th byte without any checks on the account's owner or data discriminator. It is recommended to add owner and discriminator checks, similar to how `verify_oracle_type` is implemented.

```
/* program/src/processor/oracle/refresh_oracle.rs */
024 | pub fn process_refresh_oracle(_program_id: &Pubkey, accounts: &[AccountInfo]) -> ProgramResult {
037 |     match feed.oracle_type {
038 |         0 => {
039 |             let data_source: &PullFeedAccountData = bytemuck::from_bytes(&feed_account[8..]);
066 |         }
067 |     }
072 | }
```

Although the address of the `PullFeedAccountData` account is set by the administrator, it may change in the future. Switchboard has a `pull_feed_close` instruction that can close a `PullFeedAccountData` account and reclaim it. This means the account's owner or discriminator could be modified later and no longer match what the administrator saw when set. This could open up the possibility for type confusion or data spoofing attacks.

Therefore, it is highly recommended to perform the appropriate checks every time the `PullFeedAccountData` account is used.

Resolution

The discriminator check has been added by commit `64cb105` and `e5083a2`.

COMMIT DDF437F

[P1-I-01] Redundant checks on mint

Identified in commit `ddf437f`.

There are duplicated mint checks in `PushLzBridgeAccounts:checked_from_accounts` and `PushCctpBridgeAccounts::checked_from_accounts`. For example,

```
/* program/src/integrations/cctp_bridge/push.rs */
052 | if ctx.mint.key().ne(&config.mint) {
053 |     msg! {"mint: does not match config"};
054 |     return Err(ProgramError::InvalidAccountData);
055 | }

072 | if ctx.mint.key().ne(&config.mint) {
073 |     msg! {"mint: does not match config"};
074 |     return Err(ProgramError::InvalidAccountData);
075 | }
```

Resolution

This issue has been fixed by `cd9f272` and `59adc63`.

COMMIT DDF437F

[P1-I-02] Typos

Identified in commit `ddf437f`.

1. When the`vault_b` does not match the `reserve_b.vault`, it should throw the error `ProgramError::InvalidAccountData` instead of `ProgramError::InvalidAccountOwner`.

```
/* program/src/integrations/spl_token_swap/push.rs */
166 | if inner_ctx.vault_b.key().ne(&reserve_b.vault) {
167 |     msg! {"vault_b: mismatch with reserve"};
168 |     return Err(ProgramError::InvalidAccountOwner);
169 | }
```

2. The error message for incorrect vault parameters in `process_push_cctp_bridge` is wrong.

```
/* program/src/integrations/cctp_bridge/push.rs */
143 | if inner_ctx.vault.key().ne(&reserve.vault) {
144 |     msg! {"mint: mismatch with reserve"}; // @audit: It should be "vault: mismatch with reserve"
145 |     return Err(ProgramError::InvalidAccountData);
146 | }
```

3. Change `desination` to `destination`

```
/* program/src/instructions.rs */
273 |     desination_address: Pubkey,
274 |     desination_domain: u32,
277 |     desination_address: Pubkey,

/* program/src/integrations/cctp_bridge/initialize.rs */
057 | let (desination_address, desination_domain) = match outer_args.inner_args {
059 |         desination_address,
060 |         desination_domain,
061 |     } => (desination_address, desination_domain),
078 | if remote_token_messenger.domain.ne(&desination_domain) {
079 |     msg! {"desination_domain: does not match remote_token_messenger state"};
088 | destination_address: Pubkey::from(desination_address),
089 | destination_domain: desination_domain,

/* program/src/integrations/cctp_bridge/push.rs */
138 | msg! {"desination_domain: does not match remote_token_messenger state"};

/* program/src/integrations/lz_bridge/initialize.rs */
057 | let (desination_address, destination_eid) = match outer_args.inner_args {
059 |         desination_address,
060 |         destination_eid,
061 |     } => (desination_address, destination_eid),
073 | // Check the PDA of the peer_config exists for this desination_eid
097 | destination_address: Pubkey::from(desination_address),
```

4. Change `cctp_program` to `lz_program` in `InitializeLzBridgeAccounts`

```
/* program/src/integrations/lz_bridge/initialize.rs */
035 | if !ctx.oft_store.is_owned_by(ctx.lz_program.key()) {
036 |     msg! {"oft_store: not owned by cctp_program"}; // audit: change `cctp_program` to `lz_program`
037 |     return Err(ProgramError::InvalidAccountOwner);
038 | }
039 | if !ctx.peer_config.is_owned_by(ctx.lz_program.key()) {
040 |     msg! {"peer_config: not owned by cctp_program"}; // audit: change `cctp_program` to `lz_program`
041 |     return Err(ProgramError::InvalidAccountOwner);
042 | }
```

5. The error message for L28 in `emit_event` should be changed to `InvalidSeeds`.

```
/* program/src/processor/emit_event.rs */
026 | // Validate the authority is the expected controller's PDA
027 | if authority_info.key().ne(&controller_authority) {
028 |     return Err(ProgramError::MissingRequiredSignature.into());  // Should be ProgramError::InvalidSeeds.
029 | }
031 | // The authority must be the signer
032 | if !authority_info.is_signer() {
033 |     return Err(ProgramError::MissingRequiredSignature.into());
034 | }

/* program/src/state/controller.rs */
100 | if authority_info.key().ne(&controller_authority) {
101 |     // Authority PDA was invalid
102 |     return Err(ProgramError::InvalidSeeds.into());
103 | }
```

Resolution

This issue has been fixed by `cd9f272` and `59adc63`.

COMMIT DDF437F

[P1-I-03] Wrong accounts checked in `ManagePermissionAccounts::checked_from_accounts`

Identified in commit `ddf437f`.

In the `checked_from_accounts` function for `ManagePermissionAccounts`, there appears to be an issue with the account being checked in the `if` condition on line 33.

The owner of `super_permission` is already verified by the `@owner(crate::ID)` constraint within the `ManagePermissionAccounts` definition. This makes the second part of the `&&` condition in the `if` statement always false. As a result, the check on the `permission` account in the first part of `if` becomes ineffective. This would allow the `permission` account to be any account controlled by any program.

```
/* program/src/processor/manage_permission.rs */
015 | define_account_struct! {
016 |     pub struct ManagePermissionAccounts<'info> {
017 |         payer: signer, mut; // N: anyone
018 |         controller: @owner(crate::ID);
019 |         controller_authority: empty, @owner(pinocchio_system::ID);
020 |         super_authority: signer;
021 |         super_permission: @owner(crate::ID);
022 |         authority;
023 |         permission: mut;
024 |         program_id: @pubkey(crate::ID);
025 |         system_program: @pubkey(pinocchio_system::ID);
026 |     }
027 | }
028 |
029 | impl<'info> ManagePermissionAccounts<'info> {
030 |     pub fn checked_from_accounts(accounts: &'info [AccountInfo]) -> Result<Self, ProgramError> {
031 |         let ctx = Self::from_accounts(accounts)?;
032 |         if !(ctx.permission.is_owned_by(&pinocchio_system::id()) && !ctx.permission.data_is_empty())
                      // @audit: This should check `permission`, instead of `super_permission`
033 |             && !ctx.super_permission.is_owned_by(&crate::ID)
034 |         {
035 |             return Err(ProgramError::InvalidAccountOwner);
036 |         }
037 |
038 |         Ok(ctx)
039 |     }
040 | }
```

Based on the intended logic, this check should restrict the `permission` account to be either:

1. Owned by the current program (`crate::ID`), or
2. Owned by the system program and has no data.

So an incorrect account was specified on line 33. The check should target the `permission` account instead of the `super_permission` account.

Fortunately, this does not lead to any security vulnerabilities. The subsequent processor logic will revalidate the account's owner again, both when initializing the `permission` account and before reading from it.

Resolution

This issue has been fixed by `6348093`.

COMMIT DDF437F

[P1-I-04] Ensure `input_mint != output_mint`

> *Identified in commit* `ddf437f`.

The `InitializeAtomicSwapAccounts` does not verify that the `input_token` is not equal to `output_token` when initializing `AtomicSwapConfig`.

```
/* program/src/integrations/atomic_swap/initialize.rs */
016 | define_account_struct! {
017 |     pub struct InitializeAtomicSwapAccounts<'info> {
018 |         input_mint;
019 |         output_mint;
020 |         oracle: @owner(crate::ID);
021 |     }
022 | }
```

It is recommended to add the following check:

```
    if *inner_ctx.input_mint.key() == *inner_ctx.output_mint.key() {
        return Err(ProgramError::InvalidArgument);
    }
```

Resolution

This issue has been fixed by `379e449`.

COMMIT DDF437F

[P1-I-05] Insufficient ownership check for `lp_token_account`

Identified in commit `ddf437f`.

Both `PushSplTokenSwapAccounts` and `PullSplTokenSwapAccounts` require an initialized `lp_token_account` (e.g., for balance checks). However, the current validation incorrectly permits uninitialized accounts owned by `pinocchio_system::ID`.

```
/* program/src/integrations/spl_token_swap/pull.rs */
096 | if !ctx
097 |     .lp_token_account
098 |     .is_owned_by(ctx.lp_mint_token_program.key())
099 |     && !ctx.lp_token_account.is_owned_by(&pinocchio_system::ID)
100 | {
101 |     msg! {"lp_token_account: not owned by token_program or system_program"};
102 |     return Err(ProgramError::InvalidAccountOwner);
103 | }

/* program/src/integrations/spl_token_swap/push.rs */
096 | if !ctx
097 |     .lp_token_account
098 |     .is_owned_by(ctx.lp_mint_token_program.key())
099 |     && !ctx.lp_token_account.is_owned_by(&pinocchio_system::ID)
100 | {
101 |     msg! {"lp_token_account: not owned by token_program or system_program"};
102 |     return Err(ProgramError::InvalidAccountOwner);
103 | }
```

Since the `lp_token_account` must be initialized, ownership by `pinocchio_system::ID` (indicating an uninitialized state) should be rejected:

```
if !ctx.lp_token_account.is_owned_by(ctx.lp_mint_token_program.key()) {
    msg! {"lp_token_account: not owned by token_program"};
    return Err(ProgramError::InvalidAccountOwner);
}
```

Resolution

This issue has been fixed by `06f27e3`.

COMMIT DDF437F

[P1-I-06] Inconsistent variable naming in `PullSplTokenSwapAccounts`

Identified in commit `ddf437f`.

The `process_pull_spl_token_swap` withdraws tokens from the `swap`, but its variable naming (`deposit`) is inconsistent with the functionality.

```
/* program/src/integrations/spl_token_swap/pull.rs */
334 | // Carry out the actual deposit logic
386 | let post_deposit_balance_lp = lp_token_account.amount() as u128;
394 |     .checked_sub(post_deposit_balance_lp)
398 | let post_deposit_balance_a: u64;
399 | let post_deposit_balance_b: u64;
400 | if post_deposit_balance_lp > 0 {
401 |     post_deposit_balance_a =
402 |         (swap_token_a.amount() as u128 * post_deposit_balance_lp / lp_mint_supply) as u64;
403 |     post_deposit_balance_b =
404 |         (swap_token_b.amount() as u128 * post_deposit_balance_lp / lp_mint_supply) as u64;
406 |     post_deposit_balance_a = 0u64;
407 |     post_deposit_balance_b = 0u64;
413 | if step_2_balance_a != post_deposit_balance_a {
423 | after: post_deposit_balance_a,
428 | if step_2_balance_b != post_deposit_balance_b {
438 | after: post_deposit_balance_b,
446 | state.last_balance_a = post_deposit_balance_a;
447 | state.last_balance_b = post_deposit_balance_b;
448 | state.last_balance_lp = post_deposit_balance_lp as u64;
```

Consider changing the `deposit` to `withdraw` in the variable names throughout this code section.

Resolution

This issue has been fixed by `1a6fdce`.

COMMIT DDF437F

[P1-I-07] Arbitrage opportunities due to slippage-based repayments

> *Identified in commit* `ddf437f`.

In the `atomic_swap_repay` process, users are allowed to repay `tokenB` based on the current oracle price adjusted by slippage.

```
/* program/src/integrations/atomic_swap/atomic_swap_repay.rs */
223 | // min_swap_price = oracle.value * (100-max_slippage)%
224 | let min_swap_price = oracle_price
225 |     .checked_mul(BPS_DENOMINATOR.saturating_sub(max_slippage_bps).into())
226 |     .unwrap()
227 |     .checked_div(BPS_DENOMINATOR.into())
228 |     .unwrap();
230 | if swap_price < min_swap_price {
231 |     return Err(SvmAlmControllerErrors::SlippageExceeded.into());
232 | }
```

The repayment allows users to repay `tokenB` at a discounted price, and the system does not collect any protocol fees during the process.

This creates potential arbitrage risks because users can obtain `tokenB` at a price lower than the oracle price, enabling profit extraction.

Resolution

The team acknowledged this finding.

COMMIT DDF437F

[P1-I-08] Potential DoS due to rounding issue in `refresh_rate_limit`

Identified in commit `ddf437f`.

The `sync_reserve` and `sync_integration` instructions lack access control. Consequently, any user can invoke these functions and provide the necessary account information.

```
/* program/src/processor/sync_reserve.rs */
007 | define_account_struct! {
008 |     pub struct SyncReserveAccounts<'info> {
009 |         controller: @owner(crate::ID);
010 |         controller_authority: empty, @owner(pinocchio_system::ID);
011 |         reserve: mut, @owner(crate::ID);
012 |         vault;
013 |     }
014 | }

/* program/src/processor/sync_integration.rs */
016 | define_account_struct! {
017 |     pub struct SyncIntegrationAccounts<'info> {
018 |         controller: @owner(crate::ID);
019 |         controller_authority: empty, @owner(pinocchio_system::ID);
020 |         integration: mut, @owner(crate::ID);
021 |         program_id: @pubkey(crate::ID);
022 |         @remaining_accounts as remaining_accounts;
023 |     }
024 | }
```

Both functions call the `refresh_rate_limit` method on the associated reserve or integration account. This method updates the `rate_limit_outflow_amount_available` variable by adding an `increment` based on the time elapsed since the last update. However, the increment is rounded down during calculation.

```
/* program/src/state/reserve.rs */
192 | pub fn refresh_rate_limit(&mut self, clock: Clock) -> Result<(), ProgramError> {
193 |     if self.rate_limit_max_outflow == u64::MAX
194 |         || self.last_refresh_timestamp == clock.unix_timestamp
195 |     {
196 |         () // Do nothing
197 |     } else {
198 |         let increment = (self.rate_limit_slope as u128
199 |             * clock
200 |                 .unix_timestamp
201 |                 .checked_sub(self.last_refresh_timestamp)
202 |                 .unwrap() as u128
203 |             / SECONDS_PER_DAY as u128) as u64;  // rate_limit_slope * time / 24h
204 |         self.rate_limit_outflow_amount_available = self
205 |             .rate_limit_outflow_amount_available
206 |             .saturating_add(increment)
207 |             .min(self.rate_limit_max_outflow);
208 |     }
```

```
/* program/src/state/integration.rs */
217 | pub fn refresh_rate_limit(&mut self, clock: Clock) -> Result<(), ProgramError> {
218 |     if self.rate_limit_max_outflow == u64::MAX
219 |         || self.last_refresh_timestamp == clock.unix_timestamp
220 |     {
221 |         () // Do nothing
222 |     } else {
223 |         let increment = (self.rate_limit_slope as u128
224 |             * clock
225 |                 .unix_timestamp
226 |                 .checked_sub(self.last_refresh_timestamp)
227 |                 .unwrap() as u128
228 |             / SECONDS_PER_DAY as u128) as u64;
229 |         self.rate_limit_outflow_amount_available  = self
230 |             .rate_limit_outflow_amount_available
231 |             .saturating_add(increment)
232 |             .min(self.rate_limit_max_outflow);
233 |     }
```

Since the sync functions are publicly accessible, a malicious actor could repeatedly trigger them with minimal time intervals. This would cause the rate limit to remain near its `rate_limit_max_outflow` value. If `rate_limit_max_outflow` is set to a low value, this behavior could block or delay most operations, effectively denying service.

It is recommended to store the remainder of `rate_limit_slope * time_passed % SECONDS_PER_DAY` and carry it over to the next update.

Resolution

This issue has been fixed by `75e3c56`.

COMMIT DDF437F

[P1-I-09] Unnecessary `load_and_check_mut`

Identified in commit `ddf437f`.

The function `load_and_check_mut` is used to load an account that can be modified. Internally, it deserializes the account data using `NovaAccount::deserialize`, which in turn calls `BorshDeserialize::try_from_slice`. This deserialization method is not performed in place, but creates a copy of the data. For this reason, the resulting account structure is a copy, not a direct reference to the raw data area that `account_info` points to.

Therefore, any changes made to this deserialized account will not affect the raw data pointed to by `account_info`. This makes the call to `account_info.try_borrow_mut_data()` inside `load_and_check_mut` unnecessary. A simple `account_info.try_borrow_data()` is sufficient.

With this change, the implementation of `load_and_check_mut` becomes identical to `load_and_check`, making the `load_and_check_mut` function useless. We can use `load_and_check` in all cases. The program will later correctly save the changes by serializing the modified copy and writing it back to the raw data area, so there are no issues with data persistence.

The five `load_and_check_mut` functions below can be removed, and all places that use them should be changed to use `load_and_check`.

```
/* program/src/state/controller.rs */
046 | impl Controller {
072 |     pub fn load_and_check_mut(account_info: &AccountInfo) -> Result<Self, ProgramError> {
073 |         // Ensure account owner is the program
074 |         if !account_info.is_owned_by(&crate::ID) {
075 |             return Err(ProgramError::IncorrectProgramId);
076 |         }
077 |         let controller: Self =
078 |             NovaAccount::deserialize(&account_info.try_borrow_mut_data()?).unwrap();
079 |         controller.verify_pda(account_info)?;
080 |         Ok(controller)
081 |     }

/* program/src/state/integration.rs */
078 | impl Integration {
102 |     pub fn load_and_check_mut(
103 |         account_info: &AccountInfo,
104 |         controller: &Pubkey,
105 |     ) -> Result<Self, ProgramError> {
106 |         // Ensure account owner is the program
107 |         if !account_info.is_owned_by(&crate::ID) {
108 |             return Err(ProgramError::IncorrectProgramId);
109 |         }
```

```
110 |         let integration: Self =
111 |             NovaAccount::deserialize(&account_info.try_borrow_mut_data()?).unwrap();
112 |         integration.check_data(controller)?;
113 |         integration.verify_pda(account_info)?;
114 |         Ok(integration)
115 |     }

/* program/src/state/permission.rs */
072 | impl Permission {
097 |     pub fn load_and_check_mut(
098 |         account_info: &AccountInfo,
099 |         controller: &Pubkey,
100 |         authority: &Pubkey,
101 |     ) -> Result<Self, ProgramError> {
102 |         // Ensure account owner is the program
103 |         if !account_info.is_owned_by(&crate::ID) {
104 |             return Err(ProgramError::IncorrectProgramId);
105 |         }
106 |         let permission: Self =
107 |             NovaAccount::deserialize(&account_info.try_borrow_mut_data()?).unwrap();
108 |         permission.check_data(controller, authority)?;
109 |         permission.verify_pda(account_info)?;
110 |         Ok(permission)
111 |     }

/* program/src/state/reserve.rs */
071 | impl Reserve {
095 |     pub fn load_and_check_mut(
096 |         account_info: &AccountInfo,
097 |         controller: &Pubkey,
098 |     ) -> Result<Self, ProgramError> {
099 |         // Ensure account owner is the program
100 |         if !account_info.is_owned_by(&crate::ID) {
101 |             return Err(ProgramError::IncorrectProgramId);
102 |         }
103 |         let reserve: Self = NovaAccount::deserialize(&account_info.try_borrow_mut_data()?).unwrap();
104 |         reserve.check_data(controller)?;
105 |         reserve.verify_pda(account_info)?;
106 |         Ok(reserve)
107 |     }

/* program/src/state/oracle/account.rs */
068 | impl Oracle {
101 |     pub fn load_and_check_mut(account_info: &AccountInfo) -> Result<Self, ProgramError> {
102 |         // Ensure account owner is the program
103 |         if !account_info.is_owned_by(&crate::ID) {
104 |             return Err(ProgramError::IncorrectProgramId);
105 |         }
106 |         let oracle: Self = NovaAccount::deserialize(&account_info.try_borrow_mut_data()?).unwrap();
107 |         oracle.verify_pda(account_info)?;
108 |         Ok(oracle)
109 |     }
```

Resolution

This issue has been fixed by `ee9cc81`.

COMMIT DDF437F

[P1-I-10] Unnecessary `drop()`

> *Identified in commit* `ddf437f`.

To prevent reference conflicts, the `drop()` function can be used to explicitly release references.

However, if there are no further calls to the same variable, the reference will automatically be released at the end of its lifetime, making `drop` unnecessary in such cases.

The following `drop` instances are redundant and can be safely removed:

```
/* program/src/integrations/spl_token_swap/pull.rs */
183 | drop(swap_data);
389 | drop(lp_mint);
409 | drop(swap_token_a);
410 | drop(swap_token_b);

/* program/src/integrations/spl_token_swap/push.rs */
183 | drop(swap_data);
385 | drop(lp_token_account);
407 | drop(swap_token_a);
408 | drop(swap_token_b);
```

Resolution

This issue has been fixed by `3c6ff87`.

COMMIT DDF437F

[P1-I-11] Missing signer check for `message_sent_event_data`

Identified in commit `ddf437f`.

In the `process_push -> process_push_cctp_bridge` flow, the program calls the CCTP `token-messenger-minter` to initiate a cross-chain request. This CPI requires a `message_sent_event_data` account.

According to the `token-messenger-minter`'s comments and code, this account is expected to be a signer. The `token-messenger-minter` then calls `SendMessage` from the `message-transmitter` program, which is the function that actually uses, initializes, and writes data to the `message_sent_event_data` account.

```
/* https://github.com/circlefin/solana-cctp-contracts/blob/9f8cf26/programs/token-messenger-minter/src/
   token_messenger/instructions/deposit_for_burn.rs#L45-L105 */
045 | pub struct DepositForBurnContext<'info> {
093 |     /// CHECK: Account to store MessageSent event data in. Any non-PDA uninitialized address.
094 |     #[account(mut)]
095 |     pub message_sent_event_data: Signer<'info>,
105 | }

/* https://github.com/circlefin/solana-cctp-contracts/blob/9f8cf26/programs/message-transmitter/src/
   instructions/send_message.rs#L32-L60 */
032 | pub struct SendMessageContext<'info> {
046 |     #[account(
047 |         init,
048 |         payer = event_rent_payer,
049 |         space = MessageSent::len(params.message_body.len())?,
050 |     )]
051 |     pub message_sent_event_data: Box<Account<'info, MessageSent>>,
060 | }}
```

However, we can see that our program does not check if `message_sent_event_data` is a signer. It also passes extra seeds with the CPI. This makes it possible to set `message_sent_event_data` to the `controller_authority` account. When the CPI to `token-messenger-minter` occurs, `controller_authority` becomes a signer. It is then initialized by `message-transmitter`, which changes the account's owner from the `system_program` to the `message-transmitter` program.

```
/* program/src/integrations/cctp_bridge/push.rs */
023 | define_account_struct! {
024 |     pub struct PushCctpBridgeAccounts<'info> {
            // @audit: missing check to ensure message_sent_event_data is a signer
033 |         message_sent_event_data;
039 |     }
040 | }

081 | pub fn process_push_cctp_bridge(
088 | ) -> Result<(), ProgramError> {
161 |     // Perform the CPI to deposit and burn
```

```
162 |     deposit_for_burn_cpi(
166 |         Signer::from(&[
167 |             Seed::from(CONTROLLER_AUTHORITY_SEED),
168 |             Seed::from(outer_ctx.controller.key()),
169 |             Seed::from(&[controller.authority_bump]),
170 |         ]),
171 |         outer_ctx.controller_authority,
174 |         inner_ctx.vault,
181 |         inner_ctx.message_sent_event_data,
187 |     )?;
```

Because all instructions using `controller_authority` require its owner to be the `system_program` (for example, `AtomicSwapBorrow` below), this change would make it impossible to call most key instructions with the controller, leading to a Denial of Service (DoS).

```
/* program/src/integrations/atomic_swap/atomic_swap_borrow.rs */
027 | define_account_struct! {
028 |     pub struct AtomicSwapBorrow<'info> {
030 |         controller_authority: empty, @owner(pinocchio_system::ID);
042 |     }
043 | }
```

Fortunately, after calling `deposit_for_burn_cpi`, `process_push_cctp_bridge` will also call `emit_event` to log the event. This instruction also requires the `controller_authority`'s owner to be the `system_program`, preventing the attack. In addition, calling `process_push_cctp_bridge` requires the `can_invoke_external_transfer` permission, so even if the vulnerability existed, it could not be exploited by a general user.

It is recommended to add the appropriate check in `process_push_cctp_bridge` to require `message_sent_event_data` to be a signer.

Resolution

This issue has been fixed by `2886e42`.

COMMIT DDF437F

[P1-I-12] Arbitrage enabled by improper single token withdraw parameters

Identified in commit `ddf437f`.

The permission authority specifies the input `amount_a` and `amount_b` for a single-token withdrawal in the `process_pull_spl_token_swap` instruction.

```
/* program/src/integrations/spl_token_swap/pull.rs */
140 | let (amount_a, amount_b) = match outer_args {
141 |     PullArgs::SplTokenSwap { amount_a, amount_b } => (*amount_a, *amount_b),
142 |     _ => return Err(ProgramError::InvalidAccountData),
143 | };
144 | if amount_a == 0 && amount_b == 0 {
145 |     msg! {"amount_a or amount_b must be > 0"};
146 |     return Err(ProgramError::InvalidArgument);
147 | }
```

When calling the cpi `withdraw_single_token_type_exact_amount_out`, the `maximum_pool_token_amount` is hardcoded to `u64::MAX`, disabling the slippage protection mechanism.

```
/* program/src/integrations/spl_token_swap/cpi.rs */
122 | let args_vec = WithdrawSingleTokenTypeExactAmountOutArgs {
123 |     destination_token_amount: amount,
124 |     maximum_pool_token_amount: u64::MAX,
125 | }
```

During PSM swap execution, the `withdraw_single_token_type_exact_amount_out` function supports four types of curves.

- **Constant price/redemption rate curve**. It uses a specific `token_b_price` to compute `total_value`, which determines the number of lp tokens to burn.
- **Constant product/Offset curve**. Computes the LP token burn amount based on the current `swap_source_amount` using the formula.

```
ratio = source_amount / swap_source_amount
base = 1 - ratio
root = 1 - base.sqrt()
pool_tokens = pool_supply * root
```

For constant product and offset curves, reducing `swap_source_amount` increases the number of lp tokens that need to be burned.

In PSM, `maximum_pool_token_amount` acts as slippage protection. However, setting it to `u64::MAX` disables this safeguard entirely.

```
/* program/src/processor.rs */
1264 | let pool_token_amount = burn_pool_token_amount
1265 |     .checked_add(withdraw_fee)
1266 |     .ok_or(SwapError::CalculationFailure)?;
1267 |
1268 | if to_u64(pool_token_amount)? > maximum_pool_token_amount {
1269 |     return Err(SwapError::ExceededSlippage.into());
1270 | }
```

Without slippage protection, arbitrageurs can take advantage of the following sequence, using swap `token_a` as an example.

1. The arbitrageur provides liquidity, then swaps token_b for `token_a`, reducing the pool's `swap_token_a_amount`.
2. The `process_pull_spl_token_swap` instruction executes a single-token withdrawal for token_a, which now requires more lp tokens due to the manipulated `swap_token_a_amount`.
3. The arbitrageur redeems the remaining lp tokens. Since more were burned than expected in step 2, the remaining LP tokens are now overvalued, resulting in profit.

It is recommended to allow permission authority to specify a custom `maximum_pool_token_amount` when invoking the instruction, to mitigate potential arbitrage during `process_pull_spl_token_swap`.

Resolution

This issue has been fixed by `ebdf790`.

COMMIT DDF437F

[P1-I-13] Unused accounts

Identified in commit `ddf437f`.

In the program, some accounts are passed in but are never used. Similarly, some account public keys are stored but are never referenced.

1. `swap_fee_account` in `PushSplTokenSwapAccounts`

This account is not used at all in `process_push_spl_token_swap`. It is not passed to any CPI calls, and there are no checks to validate this account in any way.

```
/* program/src/integrations/spl_token_swap/push.rs */
028 | define_account_struct! {
029 |     pub struct PushSplTokenSwapAccounts<'info> {
045 |         swap_fee_account;
046 |     }
047 | }
```

2. `lookup_table` in `Integration`

This public key is set during `process_initialize_integration` and can be modified through `process_manage_integration`, but it is never actually used anywhere in the program's logic.

```
/* program/src/state/integration.rs */
031 | pub struct Integration {
037 |     /// Address Lookup Table associated with this Integration
          /// (set to Pubkey::default() when not needed)
038 |     pub lookup_table: Pubkey,
056 | }
```

Consider removing these unnecessary accounts and their associated logic.

Resolution

This issue has been fixed by `ec9c9c5` and `04c22a0`.

COMMIT DDF437F

[P1-I-14] Increment may be truncated to a small value in `refresh_rate_limit`

Identified in commit `ddf437f`.

When the rate limit is refreshed in `refresh_rate_limit`, the calculation for the `increment` value has a potential integer overflow issue.

It is possible for the `increment` value to exceed the maximum size of a `u64` before it is cast. For an extreme example, if `rate_limit_slope` is `u64::MAX` and the time since the last refresh is more than one day, the actual calculated value for `increment` will be larger than what a `u64` can hold. Simply casting it with `as u64` will truncate the value, which could make it a very small number. This would prevent `rate_limit_outflow_amount_available` from increasing as expected.

```
/* program/src/state/reserve.rs */
071 | impl Reserve {
192 |     pub fn refresh_rate_limit(&mut self, clock: Clock) -> Result<(), ProgramError> {
193 |         if self.rate_limit_max_outflow == u64::MAX
194 |             || self.last_refresh_timestamp == clock.unix_timestamp
195 |         {
196 |             () // Do nothing
197 |         } else {
198 |             let increment = (self.rate_limit_slope as u128
199 |                 * clock
200 |                     .unix_timestamp
201 |                     .checked_sub(self.last_refresh_timestamp)
202 |                     .unwrap() as u128
203 |                 / SECONDS_PER_DAY as u128) as u64;
204 |             self.rate_limit_outflow_amount_available = self
205 |                 .rate_limit_outflow_amount_available
206 |                 .saturating_add(increment)
207 |                 .min(self.rate_limit_max_outflow);
208 |         }
212 |     }
309 | }

/* program/src/state/integration.rs */
078 | impl Integration {
217 |     pub fn refresh_rate_limit(&mut self, clock: Clock) -> Result<(), ProgramError> {
218 |         if self.rate_limit_max_outflow == u64::MAX
219 |             || self.last_refresh_timestamp == clock.unix_timestamp
220 |         {
221 |             () // Do nothing
222 |         } else {
223 |             let increment = (self.rate_limit_slope as u128
224 |                 * clock
225 |                     .unix_timestamp
226 |                     .checked_sub(self.last_refresh_timestamp)
227 |                     .unwrap() as u128
228 |                 / SECONDS_PER_DAY as u128) as u64;
229 |             self.rate_limit_outflow_amount_available  = self
230 |                 .rate_limit_outflow_amount_available
```

```
231 |                 .saturating_add(increment)
232 |                 .min(self.rate_limit_max_outflow);
233 |         }
237 |     }
278 | }
```

It is recommended to change `as u64` to `.try_into().unwrap_or(u64::MAX)`. This will use the maximum possible `u64` value as the increment when the calculated value exceeds the `u64` range, instead of truncating it.

Resolution

This issue has been fixed by `29260f5`.

COMMIT C3836E1

[P2-L-01] Missing mint extension validation in integration initialization

Identified in commit `c3836e1`.

With the newly added `token_2022` support, mint accounts are expected to have their extensions validated against the `VALID_MINT_EXTENSIONS` list.

```
/* program/src/processor/initialize_reserve.rs */
061 | // Validate the mint
062 | // Load in the mint account, validating it in the process
063 | Mint::from_account_info(ctx.mint)?;
064 | validate_mint_extensions(ctx.mint)?;
```

However, only `initialize_reserve` currently performs this check. The initialization processes for various integrations do not validate mint extensions, which could lead to cases where a corresponding reserve cannot be created.

```
/* program/src/integrations/spl_token_external/initialize.rs */
033 | // Load in the mint account, validating it in the process
034 | Mint::from_account_info(inner_ctx.mint)?;

/* program/src/integrations/spl_token_swap/initialize.rs */
085 | // Load in the mint accounts, validating it in the process
086 | Mint::from_account_info(inner_ctx.mint_a)?;
087 | Mint::from_account_info(inner_ctx.mint_b)?;
088 | let lp_mint = Mint::from_account_info(inner_ctx.lp_mint)?;

/* program/src/integrations/cctp_bridge/initialize.rs */
065 | // Load in the CCTP Local Token Account and verify the mint matches
066 | let local_token =
067 |     LocalToken::deserialize(&mut &*inner_ctx.local_token.try_borrow_data()?).map_err(|e| e)?;
068 | if local_token.mint.ne(inner_ctx.mint.key()) {
069 |     msg! {"mint: does not match local_token state"};
070 |     return Err(ProgramError::InvalidAccountData);
071 | }

/* program/src/integrations/lz_bridge/initialize.rs */
065 | // Load in the LZ OFT Store Account and verify the mint matches
066 | let oft_store =
067 |     OFTStore::deserialize(&mut &*inner_ctx.oft_store.try_borrow_data()?).map_err(|e| e)?;
068 | if oft_store.token_mint.ne(inner_ctx.mint.key()) {
069 |     msg! {"mint: does not match oft_store state"};
070 |     return Err(ProgramError::InvalidAccountData);
071 | }

/* program/src/integrations/atomic_swap/initialize.rs */
051 | let input_mint = Mint::from_account_info(inner_ctx.input_mint)?;
052 | let output_mint = Mint::from_account_info(inner_ctx.output_mint)?;
```

For `spl_token_swap` integrations, omitting mint extension validation does not pose an immediate risk be-

cause mints in PSM are already restricted by a whitelist. In other cases, however, the absence of this check could allow the creation of mints with unsupported or unsafe extensions. These mints may later be unable to create a corresponding reserve, which would prevent subsequent push or pull operations from executing.

```
/* program/src/constraints.rs */
108 | const VALID_POOL_MINT_EXTENSIONS: &[ExtensionType] = &[
109 |     ExtensionType::ConfidentialTransferMint,
110 |     ExtensionType::MetadataPointer,
111 |     ExtensionType::TokenMetadata,
112 | ];
113 |
114 | const VALID_TOKEN_A_B_EXTENSIONS: &[ExtensionType] = &[
115 |     ExtensionType::ConfidentialTransferMint,
116 |     ExtensionType::MintCloseAuthority,
117 |     ExtensionType::MetadataPointer,
118 |     ExtensionType::TokenMetadata,
119 | ];
```

It is recommended to add `validate_mint_extensions` checks in all integration initializations to prevent the creation of invalid integrations.

Resolution

This issue has been fixed by `9f1b32b`.

COMMIT C3836E1

[P2-L-02] Missing Token-2022 `Mint` and `TokenAccount` type checks

Identified in commit `c3836e1`.

The program does not perform any type checks when parsing `TokenAccount` and `Mint` accounts that belong to the Token-2022 program. This can lead to type confusion vulnerabilities.

First, it is important to note that accounts for both the original Token program and Token-2022 do not have a leading discriminator to identify the account type. For the original Token program, this is not an issue because `Mint` and `TokenAccount` have different, fixed lengths. They can be distinguished simply by checking the account's data length, which the program correctly does.

However, with Token-2022, `TokenAccount` and `Mint` accounts have variable lengths because of extensions. The program only checks if the account's data length is greater than the base size for that account type. This check is not sufficient to distinguish between a `Mint` and a `TokenAccount`. It cannot even distinguish them from a `Multisig` account, which also exists under the Token-2022 program.

In a `Multisig` account, most of the fields (except for two bytes representing `n` and `is_initialized`) can be fully controlled by the creator. An attacker could use this flaw to create a malicious `Multisig` account that passes the program's checks as if it were a valid `Mint` or `TokenAccount`, leading to a type confusion attack.

```
/* pinocchio-token-interface/src/lib.rs */
015 | impl<'info> TokenAccount<'info> {
016 |     pub fn from_account_info(account_info: &'info AccountInfo) -> Result<Self, ProgramError> {
017 |         if account_info.is_owned_by(&pinocchio_token2022::ID) {
018 |             pinocchio_token2022::state::TokenAccount::from_account_info(account_info)
019 |                 .map(|t| TokenAccount(t))
020 |                 .map_err(|_| ProgramError::InvalidAccountData)
035 |         }
036 |     }
037 | }

/* pinocchio-token2022/src/state/token.rs */
051 | impl TokenAccount {
052 |     pub const BASE_LEN: usize = core::mem::size_of::<TokenAccount>();
059 |     pub fn from_account_info(
060 |         account_info: &AccountInfo,
061 |     ) -> Result<Ref<TokenAccount>, ProgramError> {
        // @audit: only require data_len >= Self::BASE_LEN is not enough
062 |         if account_info.data_len() < Self::BASE_LEN {
063 |             return Err(ProgramError::InvalidAccountData);
064 |         }
065 |         if !account_info.is_owned_by(&ID) {
066 |             return Err(ProgramError::InvalidAccountData);
```

```
067 |         }
068 |         Ok(Ref::map(account_info.try_borrow_data()?, |data| unsafe {
069 |             Self::from_bytes_unchecked(data)
070 |         }))
071 |     }
203 | }

/* pinocchio-token-interface/src/lib.rs */
049 | impl<'info> Mint<'info> {
050 |     pub fn from_account_info(account_info: &'info AccountInfo) -> Result<Self, ProgramError> {
051 |         if account_info.is_owned_by(&pinocchio_token2022::ID) {
052 |             pinocchio_token2022::state::Mint::from_account_info(account_info)
053 |                 .map(|t| Mint(t))
054 |                 .map_err(|_| ProgramError::InvalidAccountData)
067 |         }
068 |     }
069 | }

/* pinocchio-token2022/src/state/mint.rs */
037 | impl Mint {
039 |     pub const BASE_LEN: usize = core::mem::size_of::<Mint>();
046 |     pub fn from_account_info(account_info: &AccountInfo) -> Result<Ref<Mint>, ProgramError> {
       // @audit: only require data_len >= Self::BASE_LEN is not enough
047 |         if account_info.data_len() < Self::BASE_LEN {
048 |             return Err(ProgramError::InvalidAccountData);
049 |         }
050 |         if !account_info.is_owned_by(&ID) {
051 |             return Err(ProgramError::InvalidAccountOwner);
052 |         }
053 |         Ok(Ref::map(account_info.try_borrow_data()?, |data| unsafe {
054 |             Self::from_bytes_unchecked(data)
055 |         }))
056 |     }
150 | }
```

The correct way to unpack and check these accounts should follow the implementation found in the official Token-2022 program.

```
/// all below code is from https://github.com/solana-program/token-2022/tree/2e799e3

/* program/src/extension/mod.rs */
899 | impl<'data, S: BaseState + Pod> PodStateWithExtensionsMut<'data, S> {
903 |     pub fn unpack(input: &'data mut [u8]) -> Result<Self, ProgramError> {
904 |         check_min_len_and_not_multisig(input, S::SIZE_OF)?;
905 |         let (base_data, rest) = input.split_at_mut(S::SIZE_OF);
906 |         let base = pod_from_bytes_mut::<S>(base_data)?;
907 |         if !base.is_initialized() {
908 |             Err(ProgramError::UninitializedAccount)
909 |         } else {
910 |             let (account_type, tlv_data) = unpack_type_and_tlv_data_mut::<S>(rest)?;
911 |             Ok(Self {
912 |                 base,
913 |                 account_type,
914 |                 tlv_data,
915 |             })
916 |         }
917 |     }
```

```
/* program/src/extension/mod.rs */
264 | fn check_min_len_and_not_multisig(input: &[u8], minimum_len: usize) -> Result<(), ProgramError> {
265 |     if input.len() == Multisig::LEN || input.len() < minimum_len {
266 |         Err(ProgramError::InvalidAccountData)
267 |     } else {
268 |         Ok(())
269 |     }
270 | }

/* program/src/extension/mod.rs */
989 | fn unpack_type_and_tlv_data_mut<S: BaseState>(
990 |     rest: &mut [u8],
991 | ) -> Result<(&mut [u8], &mut [u8]), ProgramError> {
992 |     unpack_type_and_tlv_data_with_check_mut::<S, _>(rest, check_account_type::<S>)
993 | }

/* program/src/extension/mod.rs */
272 | fn check_account_type<S: BaseState>(account_type: AccountType) -> Result<(), ProgramError> {
273 |     if account_type != S::ACCOUNT_TYPE {
274 |         Err(ProgramError::InvalidAccountData)
275 |     } else {
276 |         Ok(())
277 |     }
278 | }

/* program/src/extension/mod.rs */
967 | fn unpack_type_and_tlv_data_with_check_mut<
968 |     S: BaseState,
969 |     F: Fn(AccountType) -> Result<(), ProgramError>,
970 | >(
971 |     rest: &mut [u8],
972 |     check_fn: F,
973 | ) -> Result<(&mut [u8], &mut [u8]), ProgramError> {
974 |     if let Some((account_type_index, tlv_start_index)) = type_and_tlv_indices::<S>(rest)? {
975 |         // type_and_tlv_indices() checks that returned indexes are within range
976 |         let account_type = AccountType::try_from(rest[account_type_index])
977 |             .map_err(|_| ProgramError::InvalidAccountData)?;
978 |         check_fn(account_type)?;
979 |         let (account_type, tlv_data) = rest.split_at_mut(tlv_start_index);
980 |         Ok((
981 |             &mut account_type[account_type_index..tlv_start_index],
982 |             tlv_data,
983 |         ))
984 |     } else {
985 |         Ok((&mut [], &mut []))
986 |     }
987 | }

/* program/src/extension/mod.rs */
1380 | impl BaseState for PodAccount {
1381 |     const ACCOUNT_TYPE: AccountType = AccountType::Account;
1382 | }
1383 | impl BaseState for PodMint {
1384 |     const ACCOUNT_TYPE: AccountType = AccountType::Mint;
1385 | }
```

To summarize, the following conditions should be met for a valid check:

```
TokenAccount:
(
    len == TokenAccount::BASE_LEN or
    (len > TokenAccount::BASE_LEN and
        len != Multisig::LEN (355) and
        account_type == AccountType::Account == 2 (at BASE_ACCOUNT_LENGTH(165))
    )
)

Mint:
(
    len == Mint::BASE_LEN or
    (len > Mint::BASE_LEN and
        len != Multisig::LEN (355) and
        account_type == AccountType::Mint == 1 (at BASE_ACCOUNT_LENGTH(165))
    )
)
```

The actual impact of this issue in the program is low. This is because the program generally does not make critical decisions based on the state of non-ATA `TokenAccount`s or untrusted `Mint`s. The main consequence is that the accuracy of some emitted events could be affected.

Resolution

This issue has been fixed by `1dd0498`.

COMMIT C3836E1

[P2-L-03] Potential DoS caused by `PermanentDelegate` extension

Identified in commit `c3836e1`.

The program allows the use of `Mint`s that have the `PermanentDelegate` extension enabled. This means the balance in a reserve's `vault` can decrease without the program itself initiating a transfer. This can cause the `sync_balance` function to fail, leading to a widespread but temporary Denial of Service (DoS).

The `sync_balance` function is called before a reserve is used in various operations. Its purpose is to update the reserve's accounting with any funds that may have been transferred in externally.

As the code comments suggest, the program assumes that a decrease in the balance between operations is an event that should not happen. All transfers initiated by the program are properly accounted for, so any unexpected change should be an inflow.

This assumption is correct for mints under the Token program. However, with the introduction of the Token-2022 program's `PermanentDelegate` extension, an external permanent delegate can transfer tokens from any token account of that mint at any time. This can cause the amount in the `vault` to decrease unexpectedly.

When this decrease is larger than the available outflow rate limit, the call to `update_for_outflow` within `sync_balance` will fail. This failure will even block operations that are intended to deposit funds and increase the vault's balance, causing a DoS.

Since the outflow limit recovers over time, this DoS condition will eventually resolve itself.

```
/* program/src/processor/shared/token_extensions.rs */
009 | pub const VALID_MINT_EXTENSIONS: &[ExtensionType] = &[
018 |     /*
019 |         UNTESTED Could transfer/burn Controller tokens.
020 |         Necessary for a lot of RWAs. Requires
021 |         trusting of the issuer.
022 |     */
023 |     ExtensionType::PermanentDelegate, // Q: dangerous, balances could drop with no transfer
024 |     /* UNTESTED Could freeze within Controller. Requires trusting of the issuer. */
044 | ];

/* program/src/state/reserve.rs */
257 | pub fn sync_balance(
263 | ) -> Result<(), ProgramError> {
282 |     if self.last_balance != new_balance {
283 |         let previous_balance = self.last_balance;
284 |
285 |         // Update the rate limits and balance for the change
```

```
286 |         if new_balance > self.last_balance {
287 |             // => inflow
288 |             self.update_for_inflow(clock, new_balance.checked_sub(self.last_balance).unwrap())?;
289 |         } else {
290 |             // new_balance < previous_balance => outflow (should not be possible)
291 |             self.update_for_outflow(
292 |                 clock,
293 |                 self.last_balance.checked_sub(new_balance).unwrap(),
294 |             )?;
295 |         }
313 | }

/* program/src/state/reserve.rs */
240 | pub fn update_for_outflow(&mut self, clock: Clock, outflow: u64) -> Result<(), ProgramError> {
247 |     self.rate_limit_outflow_amount_available = self
248 |         .rate_limit_outflow_amount_available
249 |         .checked_sub(outflow)
250 |         .ok_or(SvmAlmControllerErrors::RateLimited)?;
251 |     self.last_balance = self.last_balance.checked_sub(outflow).unwrap();
252 |     Ok(())
253 | }
```

When `sync_balance` detects an unexpected outflow, consider at most reducing `rate_limit_outflow_amount_available` to zero, rather than reverting the transaction.

Resolution

This issue has been fixed by `b404447`.

COMMIT C3836E1

[P2-L-04] Inaccurate inflow accounting if `TransferFeeConfig` is enabled

Identified in commit `c3836e1`.

In the `process_pull_spl_token_swap` function, the input parameters `amount_a` and `amount_b` are passed to an external program (Nova PSM). This external program then transfers that amount of tokens to the reserve's `vault`.

However, the current program allows the use of `Mint`s that have the `TransferFeeConfig` extension enabled. When this is the case, the actual number of tokens received by the `vault` will be the amount transferred minus the transfer fee. This means the received amount might be slightly lower than `amount_a` or `amount_b`.

The problem is that the program records the inflow using the original `amount_a` and `amount_b` values, without accounting for the deducted fee. This leads to an inaccurate `last_balance` in the reserve, which negatively affects the flow control logic.

```
/* program/src/processor/shared/token_extensions.rs */
009 | pub const VALID_MINT_EXTENSIONS: &[ExtensionType] = &[
014 |     /* Tested for AtomicSwap and SplTokenExternal integrations */
015 |     ExtensionType::TransferFeeConfig,
044 | ];

/* program/src/integrations/spl_token_swap/pull.rs */
123 | pub fn process_pull_spl_token_swap(
131 | ) -> Result<(), ProgramError> {
          // @audit: 'amount_a' and 'amount_b' are the amounts sent.
          //         If the transfer fee is enabled, the vault receives
          //         a smaller amount (amount - fee).
334 |     if amount_a > 0 {
335 |         withdraw_single_token_type_exact_amount_out_cpi(
336 |             amount_a,
355 |         )?;
356 |     }
357 |     if amount_b > 0 {
358 |         withdraw_single_token_type_exact_amount_out_cpi(
359 |             amount_b,
378 |         )?;
379 |     }
          // @audit: but the code incorrectly uses the original amount
          //         for inflow accounting
457 |     if amount_a > 0 {
458 |         reserve_a.update_for_inflow(clock, amount_a)?;
459 |     }
460 |     if amount_b > 0 {
461 |         reserve_b.update_for_inflow(clock, amount_b)?;
462 |     }
463 |
```

```
464 |     Ok(())
465 | }
```

But now in Nova PSM, valid extensions do not include `TransferFeeConfig`, so this issue won't happen.

```
/* program/src/constraints.rs */
108 | const VALID_POOL_MINT_EXTENSIONS: &[ExtensionType] = &[
109 |     ExtensionType::ConfidentialTransferMint,
110 |     ExtensionType::MetadataPointer,
111 |     ExtensionType::TokenMetadata,
112 | ];
113 |
114 | const VALID_TOKEN_A_B_EXTENSIONS: &[ExtensionType] = &[
115 |     ExtensionType::ConfidentialTransferMint,
116 |     ExtensionType::MintCloseAuthority,
117 |     ExtensionType::MetadataPointer,
118 |     ExtensionType::TokenMetadata,
119 | ];
120 |
```

But it is still recommended to use the actual change in the account's balance as the inflow amount.

Resolution

This issue has been fixed by `0a01cd4`.

COMMIT C3836E1

[P2-I-01] `AtomicSwap` should reject mints with the `MintCloseAuthority` extension

Identified in commit `c3836e1`.

During the initialization of `AtomicSwap` integration, both `input_mint_decimals` and `output_mint_decimals` are recorded for subsequent use in `atomic_swap_repay`.

```
/* program/src/integrations/atomic_swap/initialize.rs */
054 | // Create the Config
055 | let config = IntegrationConfig::AtomicSwap(AtomicSwapConfig {
056 |     input_token: *inner_ctx.input_mint.key(),
057 |     output_token: *inner_ctx.output_mint.key(),
058 |     oracle: *inner_ctx.oracle.key(),
059 |     max_slippage_bps,
060 |     max_staleness,
061 |     input_mint_decimals: input_mint.decimals(),
062 |     output_mint_decimals: output_mint.decimals(),
063 |     expiry_timestamp,
064 |     padding: [0u8; 108],
065 | });

/* program/src/integrations/atomic_swap/atomic_swap_repay.rs */
169 | check_swap_slippage(
170 |     final_input_amount,
171 |     cfg.input_mint_decimals,
172 |     balance_b_delta,
173 |     cfg.output_mint_decimals,
174 |     cfg.max_slippage_bps,
175 |     oracle.value,
176 |     oracle.precision,
177 | )?;
```

Both the `input_mint` and `output_mint` support the token-2022 standard and allow the installation of the `MintCloseAuthority` extension.

After the `atomic_swap` initialization, either `input_mint` or `output_mint` could be closed, and a new `Mint` could then be created on the same mint account, which represents a different mint with potentially different decimals. This would cause the `check_swap_slippage` results to be inaccurate.

Fortunately, closing a `Mint` requires `mint.supply == 0`, which makes exploiting this vulnerability challenging

Consider prohibiting `input_mint` and `output_mint` from installing the `MintCloseAuthority` extension when initializing the `AtomicSwap` integration.

Resolution

The team acknowledged this finding.

COMMIT C3836E1

[P2-I-02] `MemoTransfer` extension is not a mint extension

Identified in commit `c3836e1`.

The `process_initialize_reserve` function validates the mint extensions and rejects the mint if its extensions are not in the `VALID_MINT_EXTENSIONS` list.

In `token_extensions.rs:29`, the `ExtensionType::MemoTransfer` is included.

```
/* program/src/processor/initialize_reserve.rs */
033 | pub fn process_initialize_reserve(
037 | ) -> ProgramResult {
064 |     validate_mint_extensions(ctx.mint)?;

/* program/src/processor/shared/token_extensions.rs */
049 | pub fn validate_mint_extensions(mint_acct: &AccountInfo) -> ProgramResult {
050 |     if mint_acct.is_owned_by(&pinocchio_token2022::ID)
051 |         && mint_acct.data_len() > pinocchio_token2022::state::Mint::BASE_LEN
052 |     {
053 |         let extension_types = get_all_extensions_for_mint(&mint_acct.try_borrow_data()?)?;
054 |         if extension_types
055 |             .iter()
056 |             .any(|ext| !VALID_MINT_EXTENSIONS.contains(ext))
057 |         {
058 |             msg!("Mint has an invalid extension");
059 |             return Err(SvmAlmControllerErrors::InvalidTokenMintExtension.into());
060 |         }
061 |     }
064 | }

/* program/src/processor/shared/token_extensions.rs */
009 | pub const VALID_MINT_EXTENSIONS: &[ExtensionType] = &[
028 |     /* UNTESTED */
029 |     ExtensionType::MemoTransfer,
```

However, the `ExtensionType::MemoTransfer` is an extension that applies to a `TokenAccount` and will not appear on a `Mint`.

```
/* program/src/extension/mod.rs */
1255 | pub fn get_account_type(&self) -> AccountType {
1256 |     match self {
1257 |         ExtensionType::Uninitialized => AccountType::Uninitialized,
1258 |         ExtensionType::TransferFeeConfig
1275 |         | ExtensionType::Pausable => AccountType::Mint,
1276 |         ExtensionType::ImmutableOwner
1279 |         | ExtensionType::MemoTransfer
1284 |         | ExtensionType::PausableAccount => AccountType::Account,
1291 |     }
1292 | }
```

Therefore, it can be removed from the `VALID_MINT_EXTENSIONS` list.

Resolution

This issue has been fixed by `0927b85`.

COMMIT C3836E1

[P2-I-03] Missing `MemoTransfer` extension support in `transfer_tokens`

Identified in commit `c3836e1`.

When the `destination_account` has the `MemoTransfer` extension enabled, a `memo` instruction must precede the transfer attempt; otherwise, the transaction fails.

```
/* token-2022/program/src/processor.rs */
299 | /// Processes a [`Transfer`](enum.TokenInstruction.html) instruction.
300 | pub(crate) fn process_transfer(
496 |     if memo_required(&destination_account) {
497 |         check_previous_sibling_instruction_is_memo()?;
498 |     }

/* token-2022/program/src/extension/memo_transfer/mod.rs */
042 | /// Check if the previous sibling instruction is a memo
043 | pub fn check_previous_sibling_instruction_is_memo() -> Result<(), ProgramError> {
044 |     let is_memo_program = |program_id: &Pubkey| -> bool {
045 |         program_id == &spl_memo::id() || program_id == &spl_memo::v1::id()
046 |     };
047 |     let previous_instruction = get_processed_sibling_instruction();
048 |     match previous_instruction {
049 |         Some(instruction) if is_memo_program(&instruction.program_id) => {}
050 |         _ => {
051 |             return Err(TokenError::NoMemo.into());
052 |         }
053 |     }
054 |     Ok(())
055 | }
```

However, the `transfer_tokens` function currently lacks both `memo` construction and `verification` checks for the `MemoTransfer` extension on recipient accounts. Transactions involving `recipient_token_account` with this extension enabled will consequently revert.

```
/* program/src/state/controller.rs */
186 | pub fn transfer_tokens(
187 |     &self,
188 |     controller: &AccountInfo,
189 |     controller_authority: &AccountInfo,
190 |     vault: &AccountInfo,
191 |     recipient_token_account: &AccountInfo,
192 |     mint: &AccountInfo,
193 |     amount: u64,
194 |     decimals: u8,
195 |     token_program: &Pubkey,
196 | ) -> Result<(), ProgramError> {
197 |     TransferChecked {
198 |         from: vault,
199 |         to: recipient_token_account,
200 |         mint,
201 |         authority: controller_authority,
```

```
202 |         amount,
203 |         decimals,
204 |         token_program,
205 |     }
206 |     .invoke_signed(&[Signer::from(&[
207 |         Seed::from(CONTROLLER_AUTHORITY_SEED),
208 |         Seed::from(controller.key()),
209 |         Seed::from(&[self.authority_bump]),
210 |     ])])?;
211 |     Ok(())
212 | }
```

Since only token account authorities can configure the `MemoTransfer` extension, potential exploiters are limited to existing account owners. The `transfer_tokens` function is invoked in three contexts:

1. `process_push_lz_bridge`: Administrator-controlled recipient account
2. `process_push_spl_token_external`: Administrator-configured recipient account
3. `process_atomic_swap_borrow`: User-controlled recipient account

For `process_push_lz_bridge` and `process_push_spl_token_external`, if the `recipient_token_account` incorrectly has the `MemoTransfer` extension installed, the integration's `push` instruction will fail. The administrator needs to recreate the integration with a `TokenAccount` that does not have the `MemoTransfer` extension.

For `process_atomic_swap_borrow`, since the `recipient_token_account` is fully user-controlled, users can choose to use an account without the `MemoTransfer` extension instead.

Consider implementing `MemoTransfer` verification or adding the memo instruction within `transfer_tokens` to ensure compatibility with configured recipient accounts.

Resolution

The team acknowledged this finding.

COMMIT C3836E1

[P2-I-04] Missing mint `Pausable` extension validation

Identified in commit `c3836e1`.

When parsing a `Mint` account, the program only deserializes the base `Mint` state and does not check any associated extensions.

As a result, when the `Pausable` extension is enabled but not validated, the program may continue execution until a later transfer call enforces the pause.

Consider adding a similar explicit `PausableConfig` check, ensuring that pause conditions are enforced immediately and that paused mints are detected without delay.

```
/* https://github.com/solana-program/token-2022/blob/program%40v8.0.1/program/src/processor.rs#L361-L365 */
361 | if let Ok(extension) = mint.get_extension::<PausableConfig>() {
362 |     if extension.paused.into() {
363 |         return Err(TokenError::MintPaused.into());
364 |     }
365 | }
```

Resolution

This issue has been fixed by `59adc63`.

COMMIT C3836E1

[P2-I-05] Check `payer_account_a` mint in `atomic_swap_repay`

Identified in commit `c3836e1`.

Non-ATA TokenAccounts can have their mints altered by `CloseAccount` and `InitializeAccount`.

In the `atomic_swap_repay` function, `payer_account_a` is a non-ATA TokenAccount, and it should be validated that `payer_account_a.mint == mint_a`.

Fortunately, this finding does not introduce a security risk, as modifying `payer_account_a`'s mint leads to two scenarios:

- `fake_payer_account_a.amount <= recipient_token_a_pre`: repayment proceeds via `payer_account_b` without affecting the controller;
- `fake_payer_account_a.amount > recipient_token_a_pre`: it causes `excess_token_a > 0`, triggering `TransferChecked` and reverting due to mint mismatch. Both outcomes pose no threat to the controller.

To improve code quality, it is still recommended to add a check for `payer_account_a.mint == mint_a` in `process_atomic_swap_repay`.

Resolution

The team acknowledged this finding.

COMMIT C3836E1

[P2-Q-01] Question on the `ConfidentialTransferMint` support

Identified in commit `c3836e1`.

Nova PSM supports token installation of the `ExtensionType::ConfidentialTransferMint` extension. However, the controller's `VALID_MINT_EXTENSIONS` does not allow the `reserve.mint` to install this extension.

```
/* https://github.com/keel-fi/solana-psm/blob/658951e/program/src/constraints.rs#L114-L119 */
114 | const VALID_TOKEN_A_B_EXTENSIONS: &[ExtensionType] = &[
115 |     ExtensionType::ConfidentialTransferMint,
116 |     ExtensionType::MintCloseAuthority,
117 |     ExtensionType::MetadataPointer,
118 |     ExtensionType::TokenMetadata,
119 | ];

/* program/src/processor/shared/token_extensions.rs */
009 | pub const VALID_MINT_EXTENSIONS: &[ExtensionType] = &[
010 |     /* UNTESTED Purely UI, so no negative impact on Controller */
011 |     ExtensionType::InterestBearingConfig,
012 |     /* UNTESTED Purely UI, so no negative impact on Controller */
013 |     ExtensionType::ScaledUiAmount,
014 |     /* Tested for AtomicSwap and SplTokenExternal integrations */
015 |     ExtensionType::TransferFeeConfig,
016 |     /* UNTESTED */
017 |     ExtensionType::MintCloseAuthority,
018 |     /*
019 |         UNTESTED Could transfer/burn Controller tokens.
020 |         Necessary for a lot of RWAs. Requires
021 |         trusting of the issuer.
022 |     */
023 |     ExtensionType::PermanentDelegate,
024 |     /* UNTESTED Could freeze within Controller. Requires trusting of the issuer. */
025 |     ExtensionType::Pausable,
026 |     // TODO need to handle remaining accounts to enable
027 |     // ExtensionType::TransferHook,
028 |     /* UNTESTED */
029 |     ExtensionType::MemoTransfer,
030 |     /* UNTESTED */
031 |     ExtensionType::ConfidentialMintBurn,
032 |     /* UNTESTED */
033 |     ExtensionType::MetadataPointer,
034 |     /* UNTESTED */
035 |     ExtensionType::TokenMetadata,
036 |     /* UNTESTED */
037 |     ExtensionType::GroupPointer,
038 |     /* UNTESTED */
039 |     ExtensionType::TokenGroup,
040 |     /* UNTESTED */
041 |     ExtensionType::GroupMemberPointer,
042 |     /* UNTESTED */
043 |     ExtensionType::TokenGroupMember,
044 | ];
```

This configuration conflict prevents `spl_token_swap` integration for Nova PSM swap pools where `token_`

`a_mint` or `token_b_mint` use the `ConfidentialTransferMint` extension.

Does the Nova controller need to support all Nova PSM swap pools in its design, or is `ConfidentialTransferMint` strictly prohibited?

If supporting all Nova PSM swap pools is required, please consider adding `ExtensionType::ConfidentialTransferMint` to `VALID_MINT_EXTENSIONS` if it aligns with the design requirements.

Resolution

This issue has been fixed by `59adc63`.

Appendix: Methodology and Scope of Work

Assisted by the Sec3 Scanner developed in-house, the manual audit particularly focused on the following work items:

- Check common security issues.
- Check program logic implementation against available design specifications.
- Check poor coding practices and unsafe behavior.
- The soundness of the economics design and algorithm is out of scope of this work

DISCLAIMER

The instance report ("Report") was prepared pursuant to an agreement between Coderrect Inc. d/b/a Sec3 (the "Company") and Nova (the "Client''). This Report solely includes the results of a technical assessment of a specific build and/or version of the Client's code specified in the Report ("Assessed Code") by the Company. The sole purpose of the Report is to provide the Client with the results of the technical assessment of the Assessed Code. The Report does not apply to any other version and/or build of the Assessed Code. Regardless of the contents of the Report, the Report does not (and should not be interpreted to) provide any warranty, representation or covenant that the Assessed Code: (i) is error and/or bug free, (ii) has no security vulnerabilities, and/or (iii) does not infringe any third-party rights. Moreover, the Report is not, and should not be considered, an endorsement by the Company of the Assessed Code and/or of the Client. Finally, the Report should not be considered investment advice or a recommendation to invest in the Assessed Code and/or the Client.

This Report is considered null and void if the Report (or any portion thereof) is altered in any manner.

ABOUT

The Sec3 audit team comprises a group of computer science professors, researchers, and industry veterans with extensive experience in smart contract security, program analysis, testing, and formal verification. We are also building automated security tools that incorporate static analysis, penetration testing, and formal verification.

At Sec3, we identify and eliminate security vulnerabilities through the most rigorous process and aided by the most advanced analysis tools.

For more information, check out our website and follow us on twitter.