Security Assessment Report

Wormhole Swap Layer

October 16, 2024

Summary

The Sec3 team (formerly Soteria) was engaged to conduct a thorough security analysis of the Solana implementation of the Swap Layer.

The artifact of the audit was the source code in folders "`solana`" and "`universal`", excluding tests, in https://github.com/wormholelabs-xyz/example-swap-layer.

The initial audit focused on the following versions and revealed 7 issues or questions.

program	type	commit
example-swap-layer	Solana	`9b8286d4aca6241c656d873d5eb021bcab747c13`

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

Table of Contents

Result Overview

Issue	Impact	Status
EXAMPLE-SWAP-LAYER		
[M-01] Senders in the payload-type messages can be forged	Medium	Resolved
[L-01] "initiate_swap_exact_in" can send multiple identical payloads	Low	Resolved
[I-01] Initialize "recipient_token_account" if needed	Info	Acknowledged
[I-02] Unnecessary signer for "sync_native"	Info	Resolved
[I-03] Redundant allocated space for StagedOutbound	Info	Resolved
[Q-01] Redeemers won't benefit from calling "transfer" after time limit expires	Question	Resolved
[Q-02] Incorrect recipient of the rent when closing "custody_token"	Question	Resolved

Findings in Detail

EXAMPLE-SWAP-LAYER

[M-01] Senders in the payload-type messages can be forged

The "stage_outbound" function allows non-senders to assist in sending messages, provided that the "sender_token" has already been delegated to the "program_transfer_authority".

This account is a PDA with a seed that includes the current outbound parameter hash. Therefore, delegating this account can be considered an indication that the user intends to send the outbound message.

However, the implementation does not verify the "sender_token.delegate"; it only checks the "delegated_amount" and determines if the "program_transfer_authority" has delegation rights through a transfer operation.

```
/* solana/programs/swap-layer/src/processor/stage_outbound.rs */
015 | pub struct StageOutbound<'info> {
024 |     #[account(
025 |         seeds = [
026 |             TRANSFER_AUTHORITY_SEED_PREFIX,
027 |             &keccak::hash(&args.try_to_vec()?).0,
028 |         ],
029 |         bump,
030 |         constraint = sender_token.is_some() @ SwapLayerError::SenderTokenRequired,
031 |     )]
032 |     program_transfer_authority: Option<UncheckedAccount<'info>>,

155 | pub fn stage_outbound(ctx: Context<StageOutbound>, args: StageOutboundArgs) -> Result<()> {
251 |     let sender = match sender_token {
252 |         Some(sender_token) => match (
253 |             &ctx.accounts.sender,
254 |             &ctx.accounts.program_transfer_authority,
255 |         ) {
273 |             (None, Some(program_transfer_authority)) => {
274 |                 // If the program transfer authority is used, we require that the delegated amount
275 |                 // is exactly the amount being transferred.
276 |                 require_eq!(
277 |                     sender_token.delegated_amount,
278 |                     transfer_amount,
279 |                     SwapLayerError::DelegatedAmountMismatch,
280 |                 );
281 |
282 |                 let (hashed_args, authority_bump) = last_transfer_authority_signer_seeds.unwrap();
```

```
283 |
284 |                 token_interface::transfer_checked(
285 |                     CpiContext::new_with_signer(
286 |                         src_token_program.to_account_info(),
287 |                         token_interface::TransferChecked {
288 |                             from: sender_token.to_account_info(),
289 |                             to: custody_token.to_account_info(),
290 |                             authority: program_transfer_authority.to_account_info(),
291 |                             mint: src_mint.to_account_info(),
292 |                         },
293 |                         &[&[
294 |                             crate::TRANSFER_AUTHORITY_SEED_PREFIX,
295 |                             &hashed_args,
296 |                             &[authority_bump],
297 |                         ]],
298 |                     ),
299 |                     transfer_amount,
300 |                     src_mint.decimals,
301 |                 )?;
302 |
303 |                 sender_token.owner
304 |             }
```

This logic is based on two assumptions:

1. Only the token account owner can set its delegate. If the "`program_transfer_authority`" is set as the delegate of a TokenAccount, it indicates that the user intends to send the outbound message.
2. Besides the owner, only the delegate can successfully act as the authority to perform a token transfer.

The assumption 1 is valid, as the "`Approve`" instruction can only be called by the TokenAccount owner.

```
/* https://github.com/solana-labs/solana-program-library/blob/656b65855a073dde371029cdaa71324f7cccd84f/
token/program-2022/src/processor.rs#L546-L591 */
546 | pub fn process_approve(
547 |     program_id: &Pubkey,
548 |     accounts: &[AccountInfo],
549 |     amount: u64,
550 |     expected_decimals: Option<u8>,
551 | ) -> ProgramResult {
552 |     let account_info_iter = &mut accounts.iter();
553 |
554 |     let source_account_info = next_account_info(account_info_iter)?;
561 |     let delegate_info = next_account_info(account_info_iter)?;
562 |     let owner_info = next_account_info(account_info_iter)?;
```

```
563 |     let owner_info_data_len = owner_info.data_len();
584 |
585 |     Self::validate_owner(
586 |         program_id,
587 |         &source_account.base.owner,
588 |         owner_info,
589 |         owner_info_data_len,
590 |         account_info_iter.as_slice(),
591 |     )?;
```

However, the assumption 2 may not hold in the context of Token 2022, which introduces the Permanent Delegate Extension (https://spl.solana.com/token-2022/extensions#permanent-delegate).

In addition to the regular delegate, the Permanent Delegate Account specified during the mint's creation can also perform unlimited transfers and act as the authority to transfer tokens from any TokenAccount under that mint without requiring additional approval.

```
/* https://github.com/solana-labs/solana-program-library/blob/656b65855a073dde371029cdaa71324f7cccd84f/
token/program-2022/src/processor.rs#L285-L403 */
285 | pub fn process_transfer(
286 |     program_id: &Pubkey,
287 |     accounts: &[AccountInfo],
288 |     amount: u64,
289 |     expected_decimals: Option<u8>,
290 |     expected_fee: Option<u64>,
291 | ) -> ProgramResult {

396 |     match (source_account.base.delegate, maybe_permanent_delegate) {
397 |         (_, Some(ref delegate)) if authority_info.key == delegate => Self::validate_owner(
398 |             program_id,
399 |             delegate,
400 |             authority_info,
401 |             authority_info_data_len,
402 |             account_info_iter.as_slice(),
403 |         )?,
```

Thus, the check in this case is insufficient.

Exploiting this vulnerability, attackers **cannot** directly steal tokens. However, they can send messages on the "`swap-layer`" as any sender, enabling them to forge the "`sender`" field in the "`RedeemMode::Payload`" type messages.

The attacker could then input arbitrary payload buffers and send any type or amount of tokens,

with the recipient on the target chain being unable to distinguish if the message was forged.

Potential attack steps

Here are the steps of a potential attack:

1. Attackers first decide on the "`StageOutboundArgs`" they want to forge and calculate the correct "`program_transfer_authority`" address.
2. Attackers create a Token 2022 mint (referred to as "`fake_token`") and set the mint's permanent delegate account as the calculated "`program_transfer_authority`".
3. Attackers create a TokenAccount for the sender they want to impersonate.
4. The attacker calls "`stage_outbound`" with the newly created TokenAccount as the "`sender_token`". Since the check involves "`delegated_amount`", the attacker ensures that "`args.amount_in`" is set to 0. This allows them to bypass the checks in lines 251-304 in "`stage_outbound.rs`", successfully creating a "`StagedOutbound`" account with a sender of their choosing and parameters they control (such as "`StagedRedeem.Payload`").

Although "`amount_in`" is set to "`0`" when calling "`StageOutbound`", during "`initiate_swap_exact_in`", all balances in the "`custody_token`" account are transferred to the "`src_swap_token`" account for Jupiter swaps.

The attacker, controlling the arguments for the Jupiter swap, can mint a certain amount of "`fake_token`" into the "`custody_token`" account and create a ("`fake_token`" <-> "`USDC`") pair on a DEX that Jupiter relies on (such as Raydium, which allows permissionless pair creation).

They can then call "`InitiateSwapExactIn`" to convert "`fake_token`" into "`USDC`", ensuring the "`USDC`" sent to the "`liquidity-layer`" is non-zero, thus preventing the transaction from failing due to a zero input amount.

This also allows the attacker to control the amount of funds sent to the target chain.

```
/* solana/programs/swap-layer/src/processor/initiate/swap/exact_in.rs */
123 | pub fn initiate_swap_exact_in<'a, 'b, 'c, 'info>(
124 |     ctx: Context<'a, 'b, 'c, 'info, InitiateSwapExactIn<'info>>,
125 |     instruction_data: Vec<u8>,
126 | ) -> Result<()>
127 | where
128 |     'c: 'info,
129 | {
140 |     let src_mint = &ctx.accounts.src_mint;
141 |     token_interface::transfer_checked(
142 |         CpiContext::new_with_signer(
143 |             src_token_program.to_account_info(),
144 |             token_interface::TransferChecked {
145 |                 from: custody_token.to_account_info(),
146 |                 to: ctx.accounts.src_swap_token.to_account_info(),
147 |                 authority: peer.to_account_info(),
148 |                 mint: src_mint.to_account_info(),
149 |             },
150 |             &[peer_signer_seeds],
151 |         ),
152 |         custody_token.amount,
153 |         src_mint.decimals,
154 |     )?;

156 |     let (shared_accounts_route, swap_args, _) =
157 |         JupiterV6SharedAccountsRoute::set_up(ctx.remaining_accounts, &instruction_data[..])?;

203 |     let (usdc_amount_out, src_dust) = shared_accounts_route.swap_exact_in(
204 |         swap_args,
205 |         swap_authority_seeds,
206 |         ctx.remaining_accounts,
207 |         Default::default(),
208 |     )?;

/* solana/programs/swap-layer/src/composite/mod.rs */
857 | pub fn swap_exact_in(
858 |     &self,
859 |     args: SharedAccountsRouteArgs,
860 |     signer_seeds: &[&[u8]],
861 |     account_infos: &'info [AccountInfo<'info>],
862 |     limit_amount: Option<u64>,
863 | ) -> Result<(u64, u64)> {

876 |     solana_program::program::invoke_signed(
877 |         &solana_program::instruction::Instruction {
878 |             program_id: jupiter_v6::JUPITER_V6_PROGRAM_ID,
879 |             accounts,
880 |             data: (jupiter_v6::SHARED_ACCOUNTS_ROUTE_SELECTOR, args)
881 |                 .try_to_vec()
882 |                 .unwrap(),
883 |         },
884 |         account_infos,
885 |         &[signer_seeds],
886 |     )?;
```

Finally, the sender in the "`StagedRedeem::Payload`" type "`SwapMessageV1`" sent to the destination chain will use the value from "`StageOutbound.info.sender`", giving the attacker control over the sender in this type of message.

Since the "`SwapMessageV1`" does not include the source mint information, there is no information about the "`fake_token`" in the message, making it impossible for the recipient on the target chain to detect any anomalies.

```
/* solana/programs/swap-layer/src/processor/initiate/swap/exact_in.rs */
123 | pub fn initiate_swap_exact_in<'a, 'b, 'c, 'info>(
124 |     ctx: Context<'a, 'b, 'c, 'info, InitiateSwapExactIn<'info>>,
125 |     instruction_data: Vec<u8>,
126 | ) -> Result<()>
127 | where
128 |     'c: 'info,
129 | {
190 |     let swap_msg = ctx.accounts.staged_outbound.to_swap_message_v1()?;

/* solana/programs/swap-layer/src/state/staged/outbound.rs */
097 | pub fn to_swap_message_v1(&mut self) -> Result<SwapMessageV1> {
098 |     let Self {
099 |         info,
100 |         staged_redeem,
101 |         encoded_output_token,
102 |     } = self;
103 |
104 |     let staged_redeem = std::mem::take(staged_redeem);
105 |
106 |     Ok(SwapMessageV1 {
107 |         recipient: info.recipient,
108 |         redeem_mode: match staged_redeem {
109 |             StagedRedeem::Direct => Default::default(),
110 |             StagedRedeem::Payload(buf) => RedeemMode::Payload {
111 |                 sender: info.sender.to_bytes(),
                        // audit: attacker can control sender
112 |                 buf: buf
113 |                     .try_into()
114 |                     .map_err(|_| SwapLayerError::PayloadTooLarge)?,
115 |             },
116 |             StagedRedeem::Relay {
117 |                 gas_dropoff,
118 |                 relaying_fee,
119 |             } => RedeemMode::Relay {
120 |                 gas_dropoff,
121 |                 relaying_fee: relaying_fee.try_into().unwrap(),
122 |             },
123 |         },
124 |         output_token: Readable::read(&mut &encoded_output_token[..])
125 |             .map_err(|_| SwapLayerError::InvalidOutputToken)?,
126 |     })
```

```
127 | }
```

Regarding the impact of such attacks, they primarily affect cross-chain applications using "`payload-type`" messages in the "`swap-layer`". If these applications rely on sender validation to determine the message sender and take specific actions based on the payload content, the impact could be severe.

For example, the "`swap-layer`" relies on the sender from the underlying "`liquidity-layer`" to verify that a message is from another chain's "`swap-layer`" contract. It is likely that other contracts using the wormhole infrastructure might also rely on the sender from the "`swap-layer`" for similar verifications.

Consider adding a check that verifies if "`sender_token.delegate == program_transfer_authority`", since the delegate can only be set by the "`Approve`" instruction that can only be called by the TokenAccount owner.

Resolution

Fixed by PR#86.

EXAMPLE-SWAP-LAYER

[L-01] "initiate_swap_exact_in" can send multiple identical payloads

In "`initiate_swap_exact_in`", if there are any remaining tokens after the swap is completed, they are kept in the "`custody_token`", and the corresponding "`staged_outbound`" account is not closed.

It can be used by senders to reclaim the remaining funds through the "`close_staged_outbound`" interface.

```
/* solana/programs/swap-layer/src/processor/initiate/swap/exact_in.rs */
123 | pub fn initiate_swap_exact_in<'a, 'b, 'c, 'info>(
124 |     ctx: Context<'a, 'b, 'c, 'info, InitiateSwapExactIn<'info>>,
125 |     instruction_data: Vec<u8>,
126 | ) -> Result<()>
127 | where
128 |     'c: 'info,
129 | {
290 |     // If there is residual, we keep the staged accounts open.
291 |     if src_dust > 0 {
292 |         msg!("Staged dust: {}", src_dust);
293 |
294 |         // Transfer dust back to the custody token.
295 |         token_interface::transfer_checked(
296 |             CpiContext::new_with_signer(
297 |                 src_token_program.to_account_info(),
298 |                 token_interface::TransferChecked {
299 |                     from: src_swap_token.to_account_info(),
300 |                     to: custody_token.to_account_info(),
301 |                     authority: swap_authority.to_account_info(),
302 |                     mint: src_mint.to_account_info(),
303 |                 },
304 |                 &[swap_authority_seeds],
305 |             ),
306 |             src_dust,
307 |             src_mint.decimals,
308 |         )?;
309 |     }
310 |
311 |     // Close the source swap token account.
312 |     token_interface::close_account(CpiContext::new_with_signer(
313 |         src_token_program.to_account_info(),
314 |         token_interface::CloseAccount {
315 |             account: src_swap_token.to_account_info(),
316 |             destination: payer.to_account_info(),
317 |             authority: swap_authority.to_account_info(),
318 |         },
319 |         &[swap_authority_seeds],
320 |     ))?;
321 |
322 |     if src_dust == 0 {
```

```
323 |         let prepared_by = &ctx.accounts.prepared_by;
324 |
325 |         // Close the custody token account.
326 |         token_interface::close_account(CpiContext::new_with_signer(
327 |             src_token_program.to_account_info(),
328 |             token_interface::CloseAccount {
329 |                 account: custody_token.to_account_info(),
330 |                 destination: prepared_by.to_account_info(),
331 |                 authority: peer.to_account_info(),
332 |             },
333 |             &[peer_signer_seeds],
334 |         ))?;
335 |
336 |         // Close the staged outbound account.
337 |         ctx.accounts
338 |             .staged_outbound
339 |             .close(prepared_by.to_account_info())?;
340 |     }
```

However, there is an extreme case where the value of the tokens provided by the user exceeds twice their set "`min_amount_out`" (possibly due to significant price fluctuations of the input token).

At the same time, since the swap uses parameters provided by the caller, the "`in_amount`" passed to Jupiter can be less than the actual balance in "`custody_token`", as long as the output value in USDC exceeds "`min_amount_out`".

As a result, after this "`initiate_swap_exact_in`" call, the remaining tokens in "`custody_token`" could still have a value exceeding "`min_amount_out`", which can potentially be used by a second "`initiate_swap_exact_in`" call.

This could lead to multiple cross-chain operations actually taking place.

```
/* solana/programs/swap-layer/src/processor/initiate/swap/exact_in.rs */
123 | pub fn initiate_swap_exact_in<'a, 'b, 'c, 'info>(
124 |     ctx: Context<'a, 'b, 'c, 'info, InitiateSwapExactIn<'info>>,
125 |     instruction_data: Vec<u8>,
126 | ) -> Result<()>
127 | where
128 |     'c: 'info,
129 | {

156 |     let (shared_accounts_route, swap_args, _) =
157 |         JupiterV6SharedAccountsRoute::set_up(ctx.remaining_accounts, &instruction_data[..])?;

202 |     // Execute swap. Keep in mind that exact in is not really exact in... so there may be residual.
203 |     let (usdc_amount_out, src_dust) = shared_accounts_route.swap_exact_in(
```

```
204 |         swap_args,
205 |         swap_authority_seeds,
206 |         ctx.remaining_accounts,
207 |         Default::default(),
208 |     )?;
210 |     // The `min_amount_out` should always be Some when swapping into USDC, this
211 |     // is guaranteed by the stage_outbound instruction.
212 |     require!(
213 |         usdc_amount_out >= staged_outbound.info.min_amount_out.unwrap(),
214 |         SwapLayerError::InsufficientAmountOut
215 |     );
```

Please note that two consecutive "`initiate_swap_exact_in`" calls cannot happen in a short period of time, because "`token_router::prepare_market_order`" will create an account on "`prepared_order`", which is a PDA of the swap-layer, with an address determined by the address of the "`staged_outbound`".

For the same "`staged_outbound`", a new "`token_router::prepare_market_order`" can only be initiated after the previous "`market_order`" has been processed by the "`token_router`" and the "`prepared_order`" has been closed.

Therefore, **the possibility of this undesired situation is relatively low**, unless the user fails to promptly reclaim the remaining funds through the "`close_staged_outbound`" interface.

```
/* solana/programs/swap-layer/src/processor/initiate/swap/exact_in.rs */
057 | /// CHECK: Mutable, seeds must be \["prepared-order", staged_outbound.key()\]
058 | #[account(
059 |     mut,
060 |     seeds = [
061 |         PREPARED_ORDER_SEED_PREFIX,
062 |         staged_outbound.key().as_ref(),
063 |     ],
064 |     bump,
065 | )]
066 | prepared_order: UncheckedAccount<'info>,
```

Multiple cross-chain requests have little impact on Redeem requests of types Relay and Direct. Users' assets are not significantly threatened, but they may incur multiple cross-chain fees and might need to call redeem multiple times on the target chain to extract their assets.

However, Redeem requests of type Payload could be more significantly affected. Given that smart contracts on the target chain might rely on the cross-chain payload to perform specific

operations, if a payload expected to be sent only once is actually sent multiple times, it could lead to unintended consequences.

For instance, on Solana, this could result in the creation of multiple "`staged_inbound`" addresses with identical "`recipient_payload`" content. If the target contract's design lacks deduplication mechanisms, this could cause the target chain's contract to perform repeated, unintended operations, potentially harming the user. The severity of this issue depends on the specific use case.

For example, if the target contract uses the payload message to perform various swap operations with the assets bridged in addition to the contract's local assets, repeated payloads could lead to more local assets being involved in swaps than expected, potentially causing losses for the user.

Consider adding a flag in "`StagedOutbound`" to indicate whether "`initiate_swap_exact_in`" has already been executed, preventing it from being executed multiple times.

Resolution

Fixed by PR#88.

EXAMPLE-SWAP-LAYER

[I-01] Initialize "recipient_token_account" if needed

In "complete_transfer_relay" and "complete_transfer_direct", if the recipient ATA does not exist, the relayer will need to create it in a separate instruction beforehand.

Consider adding the "init_if_needed" annotation to the "recipient_token_account".

1. complete_transfer_relay

```
/* solana/programs/swap-layer/src/processor/complete/transfer/relay.rs */
087 | #[account(
088 |     mut,
089 |     associated_token::mint = usdc,
090 |     associated_token::authority = recipient
091 | )]
092 | /// Recipient associated token account. The recipient authority check
093 | /// is necessary to ensure that the recipient is the intended recipient
094 | /// of the bridged tokens.
095 | recipient_token_account: Box<Account<'info, token::TokenAccount>>,
```

2. complete_transfer_direct

```
/* solana/programs/swap-layer/src/processor/complete/transfer/direct.rs */
039 | #[account(
040 |     mut,
041 |     associated_token::mint = common::USDC_MINT,
042 |     associated_token::authority = recipient
043 | )]
044 | /// Recipient associated token account. The recipient authority check
045 | /// is necessary to ensure that the recipient is the intended recipient
046 | /// of the bridged tokens. Mutable.
047 | recipient_token_account: Box<Account<'info, token::TokenAccount>>,
```

Resolution

Acknowledged. The off-chain relayer already handles the case where the ATA does not exist. It isn't difficult for the off-chain piece to add this instruction to the transaction.

EXAMPLE-SWAP-LAYER

[I-02] Unnecessary signer for "sync_native"

In the "stage_outbound" instruction, the "sync_native" call is signed using the peer's seed.

```
/* solana/programs/swap-layer/src/processor/stage_outbound.rs */
155 | pub fn stage_outbound(ctx: Context<StageOutbound>, args: StageOutboundArgs) -> Result<()> {
320 |                 let peer_seeds = &ctx.accounts.target_peer.seeds;
321 |                 token_interface::sync_native(CpiContext::new_with_signer(
322 |                     src_token_program.to_account_info(),
323 |                     token_interface::SyncNative {
324 |                         account: custody_token.to_account_info(),
325 |                     },
326 |                     &[&[
327 |                         Peer::SEED_PREFIX,
328 |                         &peer_seeds.chain.to_be_bytes(),
329 |                         &[peer_seeds.bump],
330 |                     ]],
331 |                 ))?;
333 |                 sender.key()
334 |             }
```

However, the "sync_native" instructions in the "spl-token" and "spl-token2022" programs do not require a signer.

```
/* https://github.com/solana-labs/solana-program-library/blob/656b65855a073dde371029cdaa71324f7cccd84f/
token/program/src/instruction.rs#L368-L378 */
374 | /// Accounts expected by this instruction:
375 | ///
376 | ///   0. `[writable]`  The native token account to sync with its underlying
377 | ///      lamports.
378 | SyncNative,

/* https://github.com/solana-labs/solana-program-library/blob/656b65855a073dde371029cdaa71324f7cccd84f/
token/program-2022/src/instruction.rs#L436-L446 */
442 | /// Accounts expected by this instruction:
443 | ///
444 | ///   0. `[writable]`  The native token account to sync with its underlying
445 | ///      lamports.
446 | SyncNative,
```

Resolution

Fixed by commit 4973563.

EXAMPLE-SWAP-LAYER

[I-03] Redundant allocated space for StagedOutbound

The comment in line 70 implies "encoded_output_token" can be "None". However, it is not an "Option" as seen in line 63. As a result, the calculated space is 1 byte bigger.

```
/* solana/programs/swap-layer/src/state/staged/outbound.rs */
060 | pub struct StagedOutbound {
061 |     pub info: StagedOutboundInfo,
062 |     pub staged_redeem: StagedRedeem,
063 |     pub encoded_output_token: Vec<u8>,
064 | }
065 |
066 | impl StagedOutbound {
067 |     const BASE_SIZE: usize = 8 // DISCRIMINATOR
068 |         + StagedOutboundInfo::INIT_SPACE
069 |         + 1 // StagedRedeem discrimant
070 |         + 1 // encoded_output_token === None
071 |         ;
```

The following test code prints the allocated size and the actual length after serialization:

```
mod test {
    use super::*;
    use anchor_lang::Discriminator;
    use common::wormhole_io::Writeable;

    #[test]
    fn size() {
        let payload = vec![0];
        let staged_redeem = StagedRedeem::Payload(payload.clone());
        let encoded_output_token = {
            let mut buf = Vec::with_capacity(1);
            OutputToken::Usdc.write(&mut buf).unwrap();
            buf
        };
        let res = StagedOutbound {
            info: StagedOutboundInfo {
                custody_token_bump: 0,
                prepared_by: Pubkey::default(),
                sender: Pubkey::default(),
                target_chain: 0,
                recipient: [0; 32],
                is_exact_in: false,
                usdc_refund_token: Pubkey::default(),
            },
            staged_redeem: staged_redeem,
            encoded_output_token: encoded_output_token.clone(),
        };
        println!("allocated size: {}",
```

```
                StagedOutbound::try_compute_size(&Some(RedeemOption::Payload(payload)),
                &Some(encoded_output_token)).unwrap());
        println!("real size: {}", StagedOutbound::DISCRIMINATOR.len() + res.try_to_vec().unwrap().len());
    }
}
```

The result:

```
allocated size: 152
real size: 151
```

Resolution

Fixed by commit a82e41e.

EXAMPLE-SWAP-LAYER

[Q-01] Redeemers won't benefit from calling "transfer" after time limit expires

In "complete_transfer_relay", if the time exceeds the "time_limit", the redeemer can call the "complete_transfer_relay" function instead of directly invoking "complete_swap_relay".

```
/* solana/programs/swap-layer/src/processor/complete/transfer/relay.rs */
023 | #[account(
024 |     constraint = {
025 |         let swap_msg = consume_swap_layer_fill.read_message_unchecked();
026 |
027 |         require_keys_eq!(
028 |             recipient.key(),
029 |             Pubkey::from(swap_msg.recipient),
030 |             SwapLayerError::InvalidRecipient
031 |         );
032 |
033 |         // Ensure that the swap time limit has been exceeded if the
034 |         // relayer is attempting to redeem an output token that is not USDC.
035 |         match swap_msg.output_token {
036 |             OutputToken::Usdc => {}
037 |             OutputToken::Gas(_) | OutputToken::Other { .. } => {
038 |                 // If the redeemer is not the recipient, handle these output tokens very
039 |                 // carefully by checking the time limits.
040 |                 if redeemer.key() != recipient.key() {
041 |                     let time_diff = Clock::get()
042 |                         .unwrap()
043 |                         .unix_timestamp
044 |                         .saturating_sub(consume_swap_layer_fill.fill.timestamp);
045 |                     let swap_time_limit = &consume_swap_layer_fill
046 |                         .source_peer
047 |                         .relay_params
048 |                         .swap_time_limit;
049 |
050 |                     match consume_swap_layer_fill.fill.fill_type {
051 |                         FillType::FastFill => {
052 |                             require!(
053 |                                 time_diff >= i64::from(swap_time_limit.fast_limit),
054 |                                 SwapLayerError::SwapTimeLimitNotExceeded
055 |                             );
056 |                         }
057 |                         FillType::WormholeCctpDeposit => {
058 |                             require!(
059 |                                 time_diff >= i64::from(swap_time_limit.finalized_limit),
060 |                                 SwapLayerError::SwapTimeLimitNotExceeded
061 |                             );
062 |                         }
063 |                         FillType::Unset => return Err(SwapLayerError::UnsupportedFillType.into()),
064 |                     }
065 |                 }
066 |             }
067 |         }
```

```
068 |
069 |         true
070 |     }
071 | )]
072 | consume_swap_layer_fill: ConsumeSwapLayerFill<'info>,
```

During this process, the "relaying_fee" is still transferred to the "fee_recipient_token" account, resulting in no profit for either the payer or the redeemer.

```
/* solana/programs/swap-layer/src/processor/complete/transfer/relay.rs */
192 | if user_amount != fill_amount {
193 |     token::transfer(
194 |         CpiContext::new_with_signer(
195 |             token_program.to_account_info(),
196 |             token::Transfer {
197 |                 from: complete_token.to_account_info(),
198 |                 to: ctx.accounts.fee_recipient_token.to_account_info(),
199 |                 authority: custodian.to_account_info(),
200 |             },
201 |             &[Custodian::SIGNER_SEEDS],
202 |         ),
203 |         fill_amount.checked_sub(user_amount).unwrap(),
204 |     )?;
205 | }
```

Therefore, users lack incentives to assist in completing the cross-chain process by performing the transfer immediately after the timeout.

Is this behavior intentional, meaning that the redeemer is not just any user after the time limit expires?

Resolution

Redeemers will not benefit from calling "transfer" after the time limit has expired. This is intentional to avoid competition with the protocol's fee recipient. While anyone can still redeem, there is no incentive to do so.

EXAMPLE-SWAP-LAYER

[Q-02] Incorrect recipient of the rent when closing "custody_token"

The "staged_custody_token" is created within the "stage_outbound" instruction, with the rent being paid by the payer who calls this instruction.

The payer is recorded in the "prepared_by" field of the "staged_outbound".

```
/* solana/programs/swap-layer/src/processor/stage_outbound.rs */
087 | #[account(
088 |     init,
089 |     payer = payer,
090 |     token::mint = src_mint,
091 |     token::authority = target_peer,
092 |     token::token_program = src_token_program,
093 |     seeds = [
094 |         crate::STAGED_CUSTODY_TOKEN_SEED_PREFIX,
095 |         staged_outbound.key().as_ref(),
096 |     ],
097 |     bump,
098 | )]
099 | staged_custody_token: Box<InterfaceAccount<'info, token_interface::TokenAccount>>,

154 |
155 | pub fn stage_outbound(ctx: Context<StageOutbound>, args: StageOutboundArgs) -> Result<()> {
339 |     ctx.accounts.staged_outbound.set_inner(StagedOutbound {
340 |         info: StagedOutboundInfo {
341 |             custody_token_bump: ctx.bumps.staged_custody_token,
342 |             prepared_by: ctx.accounts.payer.key(),
343 |             usdc_refund_token: ctx.accounts.usdc_refund_token.key(),
344 |             sender,
345 |             target_chain,
346 |             is_exact_in,
347 |             recipient,
348 |             min_amount_out,
349 |         },
350 |         staged_redeem,
351 |         encoded_output_token,
352 |     });
```

However, in the "initiate_transfer" instruction, when the "custody_token" account is closed, the rent is transferred to the payer who calls the "initiate_transfer" instruction instead of to the "staged_outbound.prepared_by".

```
/* solana/programs/swap-layer/src/processor/initiate/transfer.rs */
092 | pub fn initiate_transfer(ctx: Context<InitiateTransfer>) -> Result<()> {
100 |     let custody_token = &ctx.accounts.staged_custody_token;
166 |     token::close_account(CpiContext::new_with_signer(
167 |         ctx.accounts.token_program.to_account_info(),
168 |         token::CloseAccount {
169 |             account: custody_token.to_account_info(),
170 |             destination: ctx.accounts.payer.to_account_info(),
171 |             authority: ctx.accounts.custodian.to_account_info(),
172 |         },
173 |         &[Custodian::SIGNER_SEEDS],
174 |     ))
```

In contrast, in the "initiate_swap_exact_in" instruction, which is similar to "initiate_transfer", the rent from closing the "custody_token" account is correctly paid to the "prepared_by", i.e., the account that actually paid the rent during creation.

```
/* solana/programs/swap-layer/src/processor/initiate/swap/exact_in.rs */
012 | pub struct InitiateSwapExactIn<'info> {
019 |     #[account(
020 |         mut,
021 |         address = staged_outbound.info.prepared_by,
022 |     )]
023 |     prepared_by: UncheckedAccount<'info>,
037 |     #[account(
038 |         mut,
039 |         token::mint = src_mint,
040 |         token::authority = target_peer,
041 |         token::token_program = src_token_program,
042 |         seeds = [
043 |             crate::STAGED_CUSTODY_TOKEN_SEED_PREFIX,
044 |             staged_outbound.key().as_ref(),
045 |         ],
046 |         bump = staged_outbound.info.custody_token_bump,
047 |     )]
048 |     staged_custody_token: Box<InterfaceAccount<'info, token_interface::TokenAccount>>,

123 | pub fn initiate_swap_exact_in<'a, 'b, 'c, 'info>(
124 |     ctx: Context<'a, 'b, 'c, 'info, InitiateSwapExactIn<'info>>,
125 |     instruction_data: Vec<u8>,
126 | ) -> Result<()>
127 | where
128 |     'c: 'info,
129 | {
131 |     let custody_token = &ctx.accounts.staged_custody_token;
323 |         let prepared_by = &ctx.accounts.prepared_by;
324 |
325 |         // Close the custody token account.
326 |         token_interface::close_account(CpiContext::new_with_signer(
327 |             src_token_program.to_account_info(),
328 |             token_interface::CloseAccount {
```

```
329 |                 account: custody_token.to_account_info(),
330 |                 destination: prepared_by.to_account_info(),
331 |                 authority: peer.to_account_info(),
332 |             },
333 |             &[peer_signer_seeds],
334 |         ))?;
```

Should the rent in `"custody_token"` be returned to `"prepared_by"` in the `"initiate_transfer"` instruction too?

Resolution

Fixed by commit [fa76cce](fa76cce).

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 Wormhole Foundation (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.