hexens x 1inch

Security Review Report for 1inch

June 2025

Table of Contents

1. About Hexens

Hexens is a pioneering cybersecurity firm dedicated to establishing robust security standards for Web3 infrastructure, driving secure mass adoption through innovative protection technology and frameworks. As an industry elite experts in blockchain security, we deliver comprehensive audit solutions across specialized domains, including infrastructure security, Zero Knowledge Proof, novel cryptography, DeFi protocols, and NFTs.

Our methodology combines industry-standard security practices combined with unique methodology of two teams per audit, continuously advancing the field of Web3 security. This innovative approach has earned us recognition from industry leaders.

Since our founding in 2021, we have built an exceptional portfolio of enterprise clients, including major blockchain ecosystems and Web3 platforms.

2. Executive Summary

This audit covers 1inch Fusion+, a robust solution for secure and efficient cross-chain swaps in DeFi. It leverages an innovative architecture based on Dutch auctions and automated recovery mechanisms - all without relying on a centralized custodian.

Our security assessment spanned one week and involved a thorough review of the smart contracts intended for deployment on the Solana chain.

During the audit, we identified two high-severity vulnerabilities - one that could lead to temporary loss of access to funds, and another that could allow a malicious maker to steal user tokens. In addition, we reported one medium-severity issue, two low-severity issues, and two informational findings.

All reported issues were either addressed or acknowledged by the development team and subsequently verified by our team.

As a result, we can confidently say that the protocol's security posture and code quality have improved following our audit.

3. Security Review Details

▪ Review Led by

Trung Dinh, Lead Security Researcher

▪ Scope

The analyzed resources are located on:

🔗 https://github.com/1inch/solana-crosschain-protocol

📌 Commit: `06ef2533ab5dd451a6d61bda677d54e6e628e21e`

The issues described in this report were fixed in the following commit:

🔗 https://github.com/1inch/solana-crosschain-protocol

📌 Commit: `957245423aa4c77693547e15c7d5589301306b50`

▪ Changelog

11 June 2025	Audit start
18 June 2025	Initial report
09 July 2025	Revision received
11 July 2025	Final report

4. Severity Structure

The vulnerability severity is calculated based on two components:

1. Impact of the vulnerability
2. Probability of the vulnerability

Impact	Probability			
	Rare	Unlikely	Likely	Very likely
Low	Low	Low	Medium	Medium
Medium	Low	Medium	Medium	High
High	Medium	Medium	High	Critical
Critical	Medium	High	Critical	Critical

- **Severity Characteristics**

Smart contract vulnerabilities can range in severity and impact, and it's important to understand their level of severity in order to prioritize their resolution. Here are the different types of severity levels of smart contract vulnerabilities:

Critical

Vulnerabilities that are highly likely to be exploited and can lead to catastrophic outcomes, such as total loss of protocol funds, unauthorized governance control, or permanent disruption of contract functionality.

High

Vulnerabilities that are likely to be exploited and can cause significant financial losses or severe operational disruptions, such as partial fund theft or temporary asset freezing.

Medium	Vulnerabilities that may be exploited under specific conditions and result in moderate harm, such as operational disruptions or limited financial impact without direct profit to the attacker.
Low	Vulnerabilities with low exploitation likelihood or minimal impact, affecting usability or efficiency but posing no significant security risk.
Informational	Issues that do not pose an immediate security risk but are relevant to best practices, code quality, or potential optimizations.

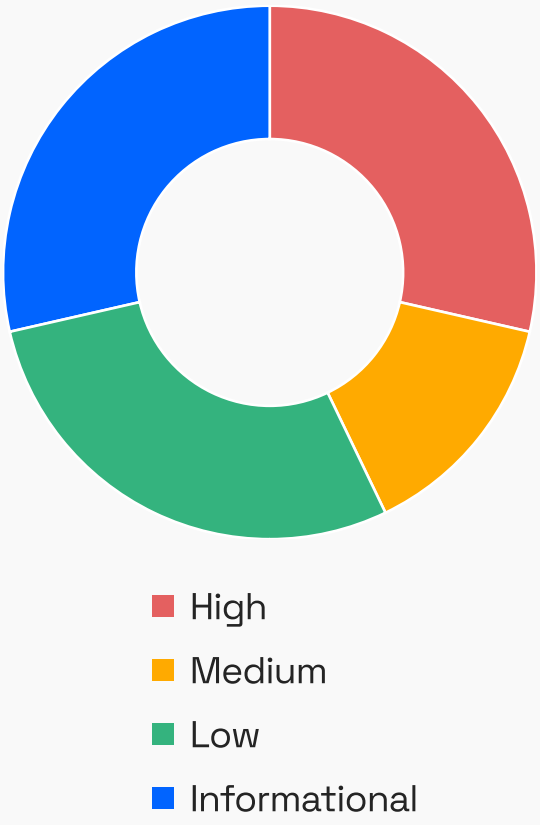
- **Issue Symbolic Codes**

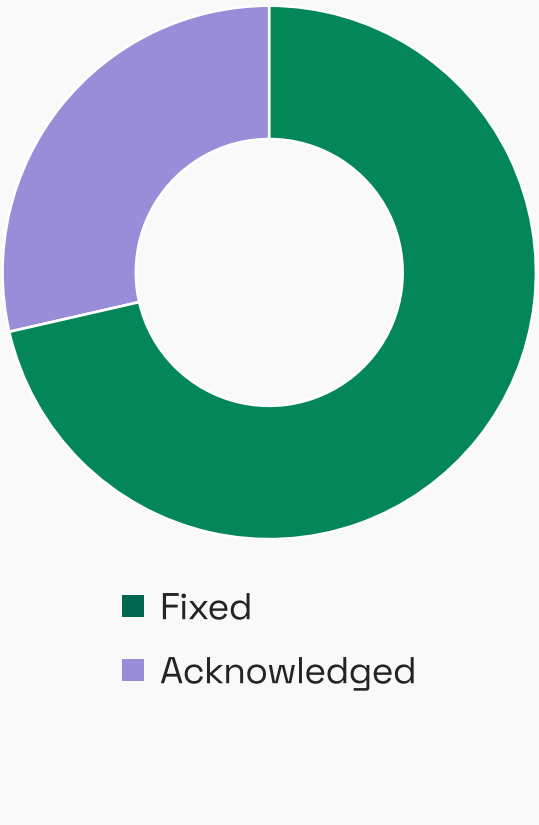
Each identified and validated issue is assigned a unique symbolic code during the security research stage.

Due to the structure of the vulnerability reporting flow, some rejected issues may be missing.

5. Findings Summary

Severity	Number of findings
Critical	0
High	2
Medium	1
Low	2
Informational	2
Total:	**7**





6. Weaknesses

This section contains the list of discovered weaknesses.

OIN9-6 | The create_escrow function sets an incorrect token amount in the escrow data

Fixed

Severity:	High	Probability:	Likely	Impact:	High

Path:

programs/cross-chain-escrow-src/src/lib.rs#L209

Description:

In the **create_escrow** function in **cross-chain-escrow-src/src/lib.rs**, the **escrow** account is initialized with the following data:

```
ctx.accounts.escrow.set_inner(EscrowSrc {
    order_hash: order.order_hash,
    hashlock,
    maker: order.creator,
    taker: ctx.accounts.taker.key(),
    token: order.token,
    amount: order.amount, // @audit: order.amount is used instead of the input
amount
    safety_deposit: order.safety_deposit,
    withdrawal_start,
    public_withdrawal_start,
    cancellation_start,
    public_cancellation_start,
    rescue_start: order.rescue_start,
    asset_is_native: order.asset_is_native,
    dst_amount: get_dst_amount(order.dst_amount, &dutch_auction_data)?,
});
```

However, using **order.amount** here is incorrect when the order allows partial fills. In such cases, the taker may initiate an escrow to fulfill only a portion of the order, represented by the **amount** input to the function. Therefore, the escrow's **amount** field should be set to the input **amount**, not **order.amount**.

This inconsistency becomes problematic because the escrow's address is derived using the input **amount**, not **order.amount**, as seen in the seeds definition for the **escrow** account in the **CreateEscrow** context:

```
#[account(
    init,
    payer = taker,
    space = constants::DISCRIMINATOR_BYTES + EscrowSrc::INIT_SPACE,
    seeds = [
        "escrow".as_bytes(),
        order.order_hash.as_ref(),
        &get_escrow_hashlock(order.hashlock, merkle_proof.clone()),
        order.creator.as_ref(),
        taker.key().as_ref(),
        mint.key().as_ref(),
        amount.to_be_bytes().as_ref(), // @audit: amount is used here
        order.safety_deposit.to_be_bytes().as_ref(),
        order.rescue_start.to_be_bytes().as_ref(),
    ],
    bump,
)]
escrow: Box<Account<'info, EscrowSrc>>,
```

Later, in functions such as **Withdraw**, the seeds used to reference the **escrow** account are reconstructed using **escrow.amount**:

```
#[account(
    mut,
    seeds = [
        "escrow".as_bytes(),
        escrow.order_hash.as_ref(),
        escrow.hashlock.as_ref(),
        escrow.maker.as_ref(),
        escrow.taker.as_ref(),
        mint.key().as_ref(),
        escrow.amount.to_be_bytes().as_ref(),
        escrow.safety_deposit.to_be_bytes().as_ref(),
        escrow.rescue_start.to_be_bytes().as_ref(),
    ],
    bump,
)]
escrow: Box<Account<'info, EscrowSrc>>,
```

Because **escrow.amount** was set to **order.amount**, but the seeds originally used amount (the actual filled amount), the derived address during withdrawal does not match the initialized escrow address. As a result, the withdrawal or cancellation function fails to locate the correct escrow account, effectively locking the funds within the escrow.

Although the funds are not permanently lost and can eventually be recovered through the **rescue_funds_for_escrow()** function, this inconsistency introduces a temporary denial of access to the taker and should be treated as a correctness and usability issue.

Remediation:

We should **not** modify the escrow's seed to use **order.amount** instead of **amount**. This is because the **escrow::close_and_withdraw_native_ata** function relies on **escrow.amount** to determine how many lamports to transfer to the recipient. If **escrow.amount** is set to **order.amount** while only a partial fill (**amount**) was actually deposited into the escrow ATA, the transfer will fail due to insufficient funds.

```
escrow.sub_lamports(escrow.amount())?;
recipient.add_lamports(escrow.amount())?;
```

Since the escrow ATA only contains **amount** worth of native tokens, attempting to transfer **order.amount** will cause an underflow and revert.

Therefore, instead of changing the seeds to use **order.amount**, we should ensure that the escrow's **amount** field is correctly set to the actual filled amount (**amount**), as shown below:

```
ctx.accounts.escrow.set_inner(EscrowSrc {
    order_hash: order.order_hash,
    hashlock,
    maker: order.creator,
    taker: ctx.accounts.taker.key(),
    token: order.token,
--  amount: order.amount,
++  amount, // Correct: use actual filled amount
    safety_deposit: order.safety_deposit,
    withdrawal_start,
    public_withdrawal_start,
    cancellation_start,
    public_cancellation_start,
    rescue_start: order.rescue_start,
    asset_is_native: order.asset_is_native,
    dst_amount: get_dst_amount(order.dst_amount, &dutch_auction_data)?,
});
```

This ensures consistency between the escrow's stored value and its actual deposited funds, and avoids transfer failures during withdrawal or rescue operations.

OIN9-10 | Order Forgery via Reused PDA Seeds Could Lead to Fund Theft

Fixed

Severity:	Probability:	Impact:
High	Unlikely	Critical

Path:

programs/cross-chain-escrow-src/src/lib.rs#L586-L602
programs/cross-chain-escrow-src/src/lib.rs#L84-L105

Description:

An order on the source chain is initialized using the following set of seeds:

```
pub struct Create<'info> {
    ...
    /// Account to store order details
    #[account(
        init,
        payer = creator,
        space = constants::DISCRIMINATOR_BYTES + Order::INIT_SPACE,
        seeds = [
            "order".as_bytes(),
            order_hash.as_ref(),
            hashlock.as_ref(),
            creator.key().as_ref(),
            mint.key().as_ref(),
            amount.to_be_bytes().as_ref(),
            safety_deposit.to_be_bytes().as_ref(),
            rescue_start.to_be_bytes().as_ref(),
        ],
        bump,
    )]
    order: Box<Account<'info, Order>>,
    ...
}
```

As shown, the seeds used to derive the order account do not include all fields from the **Order** struct -only a subset of parameters (denoted with **[x]** in the following snippet) are part of the seed derivation:

```rust
#[account]
#[derive(InitSpace)]
pub struct Order {
    order_hash: [u8; 32],              /// [x]
    hashlock: [u8; 32],                /// [x]
    creator: Pubkey,                   /// [x]
    token: Pubkey,                     /// [x]
    amount: u64,                       /// [x]
    remaining_amount: u64,
    parts_amount: u64,
    safety_deposit: u64,               /// [x]
    finality_duration: u32,
    withdrawal_duration: u32,
    public_withdrawal_duration: u32,
    cancellation_duration: u32,
    rescue_start: u32,                 /// [x]
    expiration_time: u32,
    asset_is_native: bool,
    dst_amount: [u64; 4],
    dutch_auction_data_hash: [u8; 32],
    max_cancellation_premium: u64,
    cancellation_auction_duration: u32,
    allow_multiple_fills: bool,
}
```

This partial inclusion creates a vulnerability: a malicious creator can cancel an existing order (before the resolver finalizes the escrow), and recreate a new one using the **same seeds** but with **different non-seeded parameters**, such as **withdrawal_duration**.

Consider the following scenario:

1. Alice creates Order X with **withdrawal_duration = public_withdrawal_duration = 1 day**.
2. Bob, a resolver, validates that the order has acceptable timing and prepares to take it.
3. Before Bob submits the escrow creation transaction, Alice front-runs and cancels Order X via **cross-chain-escrow-src::cancel_order()**.
4. Alice then re-creates a new order with the exact same seeds, but this time sets withdrawal_duration = public_withdrawal_duration = 0.
5. Bob proceeds with cross-chain-escrow-src::create_escrow() assuming the order is unchanged. However, the escrow is now initialized with invalid timing:
    - public_withdrawal_start = withdrawal_start = cancellation_start = public_cancellation_start

As a result, withdrawal and public withdrawal will always fail, due to these checks:

```
    pub fn withdraw(ctx: Context<Withdraw>, secret: [u8; 32]) -> Result<()> {
        let now = utils::get_current_timestamp()?;

        require!(
            now >= ctx.accounts.escrow.withdrawal_start()
                && now < ctx.accounts.escrow.cancellation_start(),
            EscrowError::InvalidTime
        );

        ...
    }
    pub fn public_withdraw(ctx: Context<PublicWithdraw>, secret: [u8; 32]) ->
Result<()> {
        let now = utils::get_current_timestamp()?;

        require!(
            now >= ctx.accounts.escrow.public_withdrawal_start()
                && now < ctx.accounts.escrow.cancellation_start(),
            EscrowError::InvalidTime
        );

        ...
    }
```

6. According to the documentation on the Withdrawal Phase:

"The 1inch relayer service ensures that both escrows, containing the required token and amount, are created, and the finality lock has passed, and then discloses the secret to all resolvers."

This means any resolver with permission can execute the withdrawal or cancellation, and they are incentivized to do so by the safety deposit.

As a result:

- On the source chain, the taker cannot withdraw their tokens, while the maker (acting maliciously) can simply wait for a resolver to call **public_cancel_escrow()** and reclaim their tokens.
- On the destination chain, even if the taker refuses to execute **withdraw()** (to release funds to the maker), another resolver can still be incentivized to call **public_withdraw()** and complete the transfer.

This opens the door for the maker (attacker) to steal the taker's funds without contributing anything.

Remediation:

Consider including finality_duration, withdrawal_duration, public_withdrawal_duration, cancellation_duration, expiration_duration, dst_amount, dutch_auction_data_hash, max_cancellation_premium, cancellation_auction_duration to the order account's seeds.

OIN9-7 | An attacker can pre-create the ATA before the escrow or order is initialized to cause a DoS in escrow creation

Fixed

Severity:	Probability:	Impact:
Medium	Unlikely	Medium

Path:

programs/cross-chain-escrow-src/src/lib.rs#L688-L695
programs/cross-chain-escrow-src/src/lib.rs#L604-L611

Description:

In the **cross-chain-escrow-src::create_escrow()** function, the **escrow_ata** is initialized to receive tokens from the **order_ata**:

```
#[derive(Accounts)]
#[instruction(amount: u64, dutch_auction_data: AuctionData, merkle_proof:
Option<MerkleProof>)]
pub struct CreateEscrow<'info> {
    ...

    #[account(
        init,
        payer = taker,
        associated_token::mint = mint,
        associated_token::authority = escrow,
        associated_token::token_program = token_program
    )]
    escrow_ata: Box<InterfaceAccount<'info, TokenAccount>>,

    ...
}
```

Here, the **escrow_ata** is created using the **init** constraint. If the associated token account already exists, the transaction will fail.

This allows an attacker to intentionally pre-create the ATA using the same **mint** and **escrow** PDA as authority, causing the escrow creation to consistently fail and resulting in a DoS condition.

A similar vulnerability exists in the order creation flow.

Remediation:

This attack vector can be mitigated by changing the **init** constraint to **init_if_needed**.

OIN9-8 | The order and escrow are not closed after the escrow is rescued

Acknowledged

Severity:	Probability:	Impact:
Low	Unlikely	Low

Path:

common/src/escrow.rs#L262-L273

Description:

Within the function **common::rescue_funds()**, if the **rescue_amount** is equal to the token amount held in **escrow_ata.amount**, the corresponding **escrow_ata** will be closed to return the rent to the **recipient**.

However, when the **escrow_ata** is closed, the corresponding **escrow** account is not. This results in unnecessary rent being held in the unused **escrow** account.

```
if rescue_amount == escrow_ata.amount {
    // Close the escrow_ata account
    close_account(CpiContext::new_with_signer(
        token_program.to_account_info(),
        CloseAccount {
            account: escrow_ata.to_account_info(),
            destination: recipient.to_account_info(),
            authority: escrow.to_account_info(),
        },
        &[seeds],
    ))?;
}
```

Remediation:

Consider closing the **escrow** if it hasn't been closed yet when the **escrow_ata** is closed.

Commentary from the client:

"We expect both the escrow and order PDAs to be closed by the time of the fund rescue."

OIN9-12 | Missing validation of the mint account in the PublicCancelEscrow context

Fixed

Severity:	Probability:	Impact:
Low	Rare	Medium

Path:

programs/cross-chain-escrow-src/src/lib.rs#L877
programs/cross-chain-escrow-src/src/lib.rs#L819
programs/cross-chain-escrow-src/src/lib.rs#L969

Description:

In the **cross-chain-escrow-src::PublicCancelEscrow** struct, the **mint** account is introduced without verifying that it matches the **token** field stored in the associated **escrow** account.

```
pub struct PublicCancelEscrow<'info> {
    ...

    mint: Box<InterfaceAccount<'info, Mint>>,

    ...

    #[account(
        mut,
        seeds = [
            "escrow".as_bytes(),
            escrow.order_hash.as_ref(),
            escrow.hashlock.as_ref(),
            escrow.maker.as_ref(),
            taker.key().as_ref(),
            escrow.token.key().as_ref(), // @audit escrow.token is used here
instead of mint
            escrow.amount.to_be_bytes().as_ref(),
            escrow.safety_deposit.to_be_bytes().as_ref(),
            escrow.rescue_start.to_be_bytes().as_ref(),
        ],
        bump,
    )]
    escrow: Box<Account<'info, EscrowSrc>>,
```

```
    ...
}
```

This lack of validation introduces a security risk: a malicious resolver can supply a **mint** account different from the one actually used in the escrow, potentially tricking the program into canceling the escrow with the wrong token - leading to fund loss for the taker.

Consider the following example:

1. Alice (taker) fills an order and creates an escrow holding 1000 USDC.
2. Once the **escrow.public_cancellation_start** time is reached, Bob (a resolver) initiates an exploit.
3. Bob:
   - Creates a new mint called HEXENS.
   - Creates a token account (ATA) for Alice's escrow PDA using HEXENS - call this **hexens_ata**.
   - Mints 1000 HEXENS into **hexens_ata**.
4. Bob then invokes **public_cancel_escrow()** using:
   - Alice's legitimate **escrow** account.
   - But supplies HEXENS as the **mint**.

Since the escrow account is the authority over **hexens_ata**, the transfer executes - but transfers HEXENS to Alice instead of USDC. Consequently Alice loses her USDC, while Bob burns a valueless token to satisfy the cancellation conditions.

Remediation:

Consider using mint as the seeds of the account **escrow** instead of **escrow.token**

```
pub struct PublicCancelEscrow<'info> {
    ...

    mint: Box<InterfaceAccount<'info, Mint>>,

    ...

    #[account(
        mut,
        seeds = [
            "escrow".as_bytes(),
            escrow.order_hash.as_ref(),
            escrow.hashlock.as_ref(),
            escrow.maker.as_ref(),
            taker.key().as_ref(),
--          escrow.token.key().as_ref(),
++          mint.key().as_ref(),
            escrow.amount.to_be_bytes().as_ref(),
            escrow.safety_deposit.to_be_bytes().as_ref(),
            escrow.rescue_start.to_be_bytes().as_ref(),
        ],
        bump,
    )]
    escrow: Box<Account<'info, EscrowSrc>>,

    ...
}
```

The fix should be applied to the struct **CancelEscrow** and **CancelOrderbyResolver** either.

OIN9-9 | Inconsistent Order State Management in Partial Rescue Operations

Fixed

Severity:	Probability:	Impact:
Informational	Rare	Medium

Description:

The **rescue_funds_for_order** function allows partial rescue of tokens from order accounts but has several state management issues:

1. Partial Rescue Without State Update: The function accepts a **rescue_amount** parameter enabling partial token rescue, but does not update **order.remaining_amount** to reflect the reduced balance. This creates a mismatch between the recorded remaining amount and actual tokens in **order_ata**.
2. Missing Time Constraint: There is no validation ensuring **rescue_start** occurs after **order.expiration_time**. This allows rescue operations during periods when **create_escrow** is still valid, potentially creating race conditions.

These issues can result in order state inconsistency, failed transactions due to insufficient balances, and potential user fund loss when escrow creation succeeds with incorrect amounts.

```
    pub fn rescue_funds_for_order(
        ctx: Context<RescueFundsForOrder>,
        order_hash: [u8; 32],
        hashlock: [u8; 32],
        order_creator: Pubkey,
        order_mint: Pubkey,
        order_amount: u64,
        safety_deposit: u64,
        rescue_start: u32,
        rescue_amount: u64,
    ) -> Result<()> {
        let seeds = [
            "order".as_bytes(),
            order_hash.as_ref(),
            hashlock.as_ref(),
            order_creator.as_ref(),
            order_mint.as_ref(),
            &order_amount.to_be_bytes(),
            &safety_deposit.to_be_bytes(),
            &rescue_start.to_be_bytes(),
            &[ctx.bumps.order],
```

```
        ];

        common::escrow::rescue_funds(
            &ctx.accounts.order,
            rescue_start,
            &ctx.accounts.order_ata,
            &ctx.accounts.resolver,
            &ctx.accounts.resolver_ata,
            &ctx.accounts.mint,
            &ctx.accounts.token_program,
            rescue_amount,
            &seeds,
        )
    }
}
```

Remediation:

- Update order state after partial rescue
- Add time validation in order creation
- Include actual balance validation in **create_escrow**

OIN9-11 | rescue_funds Does Not Support Native Escrow

Acknowledged

Severity:	Probability:	Impact:
Informational	Unlikely	Informational

Path:

common/src/escrow.rs#L249-L260

Description:

The function **escrow::rescue_funds()** always calls **uni_transfer()** with the **TokenTransfer** variant, meaning it only supports SPL token transfers and not native SOL transfers.

```
pub fn rescue_funds<'info>(
    escrow: &AccountInfo<'info>,
    rescue_start: u32,
    escrow_ata: &InterfaceAccount<'info, TokenAccount>,
    recipient: &AccountInfo<'info>,
    recipient_ata: &InterfaceAccount<'info, TokenAccount>,
    mint: &InterfaceAccount<'info, Mint>,
    token_program: &Interface<'info, TokenInterface>,
    rescue_amount: u64,
    seeds: &[&[u8]],
) -> Result<()> {
    ...

    // Transfer tokens from escrow to recipient
    uni_transfer(
        &UniTransferParams::TokenTransfer {
            from: escrow_ata.to_account_info(),
            to: recipient_ata.to_account_info(),
            authority: escrow.to_account_info(),
            mint: mint.clone(),
            amount: rescue_amount,
            program: token_program.clone(),
        },
        Some(&[seeds]),
    )?;

    ...
}
```

Remediation:

Consider adding support for native SOL transfers within **rescue_funds()**. This would be especially helpful for users who create escrows on the destination chain. When **escrow_ata** is created there, its WSOL balance may not be synced with its native SOL balance. Therefore, resolvers must explicitly call **sync_native** before being able to redeem WSOL from the **escrow_ata**.

Commentary from the client:

"Currently, rescue_funds supports only wrapped SOL for partial rescues. If additional lamports remain on the ATA, they can only be recovered by closing ATA. We are comfortable with this limitation for now."


hexens
x
1inch