hexens x 1inch

Security Review Report for 1inch

July 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 focused on updates to the Solana smart contracts of 1inch Fusion+, a secure and efficient cross-chain swap protocol. Fusion+ introduces a novel architecture built around Dutch auctions and automated recovery mechanisms, eliminating the need for centralized custodians.

Our security review was conducted over the course of one week and included a comprehensive assessment of all contracts modified in this update.

During the audit, we identified one critical-severity vulnerability that could enable a griefing attack, potentially allowing an attacker to cause the maker on the source chain to lose funds. Additionally, we uncovered two medium-severity issues and four informational findings.

All reported issues have been either resolved or acknowledged by the development team, with subsequent verification performed by our auditors.

Overall, the security posture and code quality of the protocol have improved as a result of this 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: `f9268b669a666bd73520cd790bfe6741363e038b`

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

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

📌 Commit: `b3ecfcd2675aaad5fb8d7a894a3747725ffda452`

📌 Commit: `09c6eb3b43f234fe303e32520feba2b1091018fa`

📌 Commit: `cd978f714a818e6365c6ffa1196c2f637e488faf`

📌 Commit: `82d658223cd0fbd5fa560a545a974c380435fe32`

▪ Changelog

21 July 2025	Audit start
28 July 2025	Initial report
01 August 2025	Revision received
05 August 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.

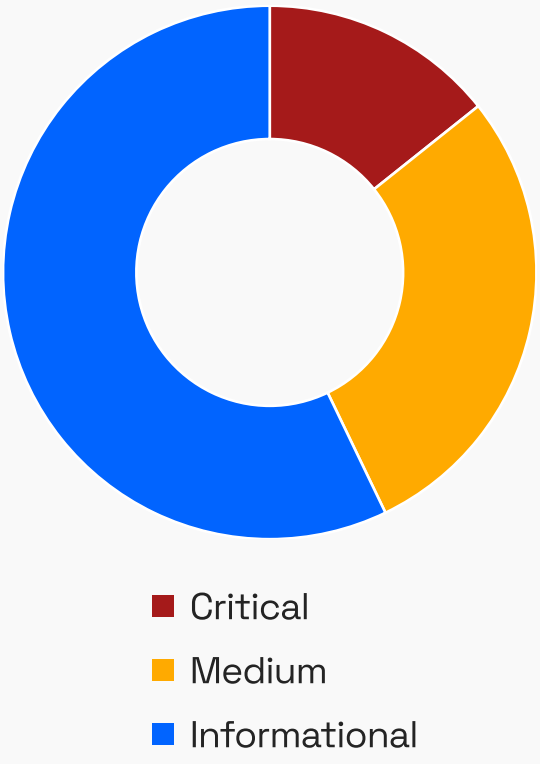
- **Issue Symbolic Codes**

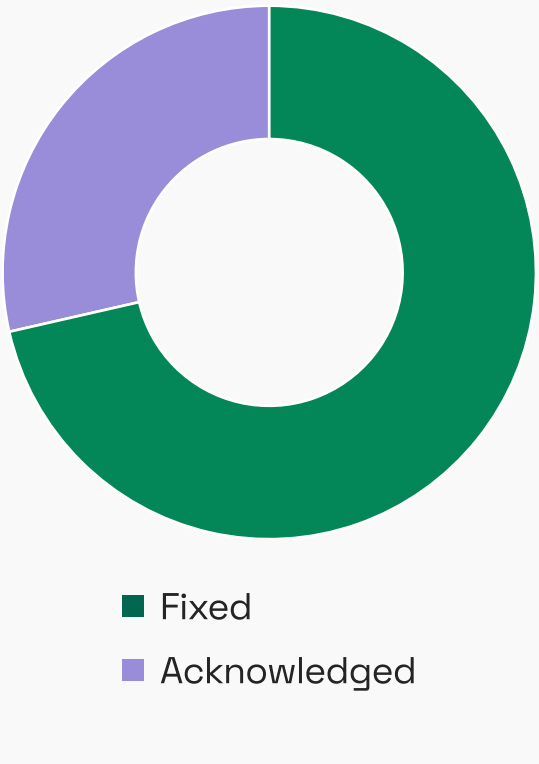
Each identified and validated issue is assigned a unique symbolic code during the security research stage.

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

5. Findings Summary

Severity	Number of findings
Critical	1
High	0
Medium	2
Low	0
Informational	4
Total:	**7**





6. Weaknesses

This section contains the list of discovered weaknesses.

OIN10-1 | Griefing Attack via Missing timelocks and mint in Escrow PDA Seeds on Destination Chain

Fixed

Severity:	Probability:	Impact:
Critical	Likely	Critical

Path:

programs/cross-chain-escrow-dst/src/lib.rs#L235-L283

Description:

By examining the **Create** context struct in **cross-chain-escrow-dst > src > lib.rs**, which is used to initialize the escrow account on the destination chain, we note two important points about the **escrow** account:

- The account can be created by anyone, not necessarily the taker, as long as they have sufficient funds
- The **timelocks** and **mint** input are not included in the PDA seed construction for the **escrow** account

```
#[derive(Accounts)]
#[instruction(order_hash: [u8; 32], hashlock: [u8; 32], amount: u64,
safety_deposit: u64, recipient: Pubkey)]
pub struct Create<'info> {
    /// Puts tokens into escrow
    #[account(
        mut, // Used to transfer lamports for native tokens and to pay for
escrow creation
    )]
    creator: Signer<'info>,

    ...

    #[account(
        init,
        payer = creator,
```

```
            space = constants::DISCRIMINATOR_BYTES + EscrowDst::INIT_SPACE,
            seeds = [
                "escrow".as_bytes(),
                order_hash.as_ref(),
                hashlock.as_ref(),
                recipient.as_ref(),
                amount.to_be_bytes().as_ref(),
            ],
            bump,
        )]
        escrow: Box<Account<'info, EscrowDst>>,

        ...
    }
```

Attack Scenario (Griefing via Escrow Hijack)

These two design choices allow for a griefing attack, in which an attacker can preemptively create a destination escrow and cause the maker to lose funds. Here's how the attack unfolds:

1. The **maker** creates an order to swap 100 USDC from chain A (source) to 100 USDC on chain B (destination).
2. The **taker** fulfills the order by creating the escrow on chain A.
3. An **attacker** observes the event and quickly calls **cross-chain-escrow-dst::create()** on chain B - but with a different **timelock** value. Since **timelock** is not part of the PDA seeds, the resulting escrow account address is still the same.
    - The attacker sets **dst_cancellation = 0**, allowing themselves to cancel the escrow at any time.
4. When the **taker** later tries to create the escrow on chain B, the transaction fails because the account at that PDA address already exists - created by the attacker.
5. After the lock periods on both chains expire, the **relayer**, seeing that both escrows exist (on chain A by the taker, and on chain B by the attacker), shares the secret with all resolvers.
6. The **attacker** immediately cancels their escrow on chain B and retrieves their funds.
7. On chain A, the **taker** or another resolver, incentivized by the safety deposit, completes the order and transfers the maker's funds to the taker.

As the consequence, the maker loses their funds on the source chain, while the destination funds (on chain B) are returned to the attacker, breaking the cross-chain atomicity and enabling a griefing exploit.

Note: A similar attack arises when creating an escrow on the destination chain without including the mint in the escrow's seeds. An attacker can forge a zero-value mint and use it in place of the correct one when calling the create() function. Since the mint is not part of the seeds, the resulting escrow address remains the same as the intended one. This allows the

attacker to deposit fake assets into the escrow on the destination chain, tricking the protocol into thinking the maker has been paid - while the taker can still claim legitimate funds on the source chain.

```
#[derive(Accounts)]
#[instruction(order_hash: [u8; 32], hashlock: [u8; 32], amount: u64,
safety_deposit: u64, recipient: Pubkey)]
pub struct Create<'info> {
    /// Puts tokens into escrow
    #[account(
        mut, // Needed because this account transfers lamports if the token is
native and to pay for the order creation
    )]
    creator: Signer<'info>,
    /// CHECK: check is not necessary as token is only used as a constraint to
creator_ata and escrow_ata
    mint: Box<InterfaceAccount<'info, Mint>>,
    #[account(
        mut,
        associated_token::mint = mint,
        associated_token::authority = creator,
        associated_token::token_program = token_program
    )]
    /// Account to store creator's tokens (Optional if the token is native)
    creator_ata: Option<Box<InterfaceAccount<'info, TokenAccount>>>,
    /// Account to store escrow details
    #[account(
        init,
        payer = creator,
        space = constants::DISCRIMINATOR_BYTES + EscrowDst::INIT_SPACE,
        seeds = [
            "escrow".as_bytes(),
            order_hash.as_ref(),
            hashlock.as_ref(),
            recipient.as_ref(),
            amount.to_be_bytes().as_ref(),
            ],
        bump,
    )]
    escrow: Box<Account<'info, EscrowDst>>,
    /// Account to store escrowed tokens
    #[account(
        init_if_needed,
```

```
        payer = creator,
        associated_token::mint = mint,
        associated_token::authority = escrow,
        associated_token::token_program = token_program
    )]
    escrow_ata: Box<InterfaceAccount<'info, TokenAccount>>,

    #[account(address = ASSOCIATED_TOKEN_PROGRAM_ID)]
    associated_token_program: Program<'info, AssociatedToken>,
    token_program: Interface<'info, TokenInterface>,
    rent: Sysvar<'info, Rent>,
    system_program: Program<'info, System>,
}
```

Remediation:

Consider including the **timelocks** and **mint** in the escrow's seeds and making sure that the taker is the only one who can creates the escrow on source chain.

OIN10-2 | Missing Timelock Ordering Validation Allows Malicious Order Creation

Acknowledged

Severity:	Probability:	Impact:
Medium	Rare	High

Path:

programs/cross-chain-escrow-dst/src/lib.rs#L110-L160

Description:

According to the documented timelock sequence:

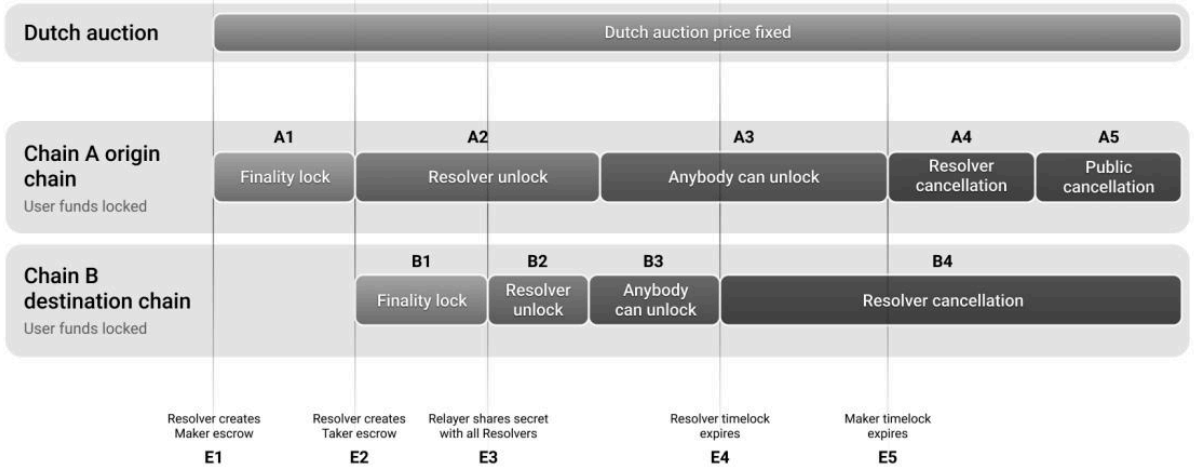


Figure 4: Timelocks sequences

the start times for each protocol phase on the source chain must satisfy the following order:

- withdrawal_start < public_withdrawal_start < cancel_escrow_start < public_cancel_escrow_start

However, when creating an order or escrow, there is **no validation to enforce this ordering**. As a result, a malicious maker can craft an order with invalid timelock sequences, potentially leading to a **loss for the taker**.

Consider the following scenario:

1. The maker creates an order with withdraw_start and public_withdraw_start greater than cancel_start
2. The taker fulfills the order and creates the corresponding escrow on the source chain
3. After the lock periods expire, the taker attempts to withdraw funds using cross-chain-escrow-src::withdraw() or public_withdraw(). These functions enforce the condition:

```
pub fn withdraw(ctx: Context<Withdraw>, secret: [u8; 32]) -> Result<()> {
    let now = get_current_timestamp()?;
    let timelocks = Timelocks(U256(ctx.accounts.escrow.timelocks));
    require!(
        now >= timelocks.get(Stage::DstWithdrawal)?
            && now < timelocks.get(Stage::DstCancellation)?,
        EscrowError::InvalidTime
    );

    ...
}

pub fn public_withdraw(ctx: Context<PublicWithdraw>, secret: [u8; 32]) ->
Result<()> {
    let now = get_current_timestamp()?;
    let timelocks = Timelocks(U256(ctx.accounts.escrow.timelocks));
    require!(
        now >= timelocks.get(Stage::DstPublicWithdrawal)?
            && now < timelocks.get(Stage::DstCancellation)?,
        EscrowError::InvalidTime
    );

    ...
}
```

But since the maker set **withdrawal_start** after **cancellation_start**, this check fails, and the withdrawal reverts.

4. As a result:

- The taker cannot access the maker's funds on the source chain.
- The maker still receives funds on the destination chain, either by taker's withdrawal or via resolvers who are incentivized by the safety deposit.

Remediation:

Consider validating the timelocks order when creating a new order.

Commentary from the client:

"The backend will validate order parameters. Orders with malicious data will not be tracked or shared with resolvers."

OIN10-3 | Missing Overflow Check on Timelocks Allows Escrow Lockout

Acknowledged

Severity:	Probability:	Impact:
Medium	Unlikely	Medium

Path:

common/src/timelocks.rs#L40-L48

Description:

The function **cross-chain-escrow-src::cancel_escrow()** is designed to allow cancellation of an escrow if the current timestamp exceeds the cancellation phase's start time. It retrieves this start time by calling:

```
Timelocks(U256(ctx.accounts.escrow.timelocks)).get(Stage::SrcCancellation)?
```

The underlying implementation of **timelocks::get()** is:

```
pub fn get(self, stage: Stage) -> Result<u32, ProgramError> {
    let shift = (stage as usize) * STAGE_BIT_SIZE;
    let deployed_at = (self.0 >> DEPLOYED_AT_OFFSET).as_u32();
    let delta = ((self.0 >> shift) & U256::from(u32::MAX)).as_u32();
    let result = deployed_at
        .checked_add(delta)
        .ok_or(ProgramError::ArithmeticOverflow)?;
    Ok(result)
}
```

This logic fails with **ProgramError::ArithmeticOverflow** if the sum of **deployed_at** and **delta** exceeds the **u32** limit. For example, setting **delta = u32::MAX** will cause the function to revert.

A malicious maker can exploit this by setting an unreasonably large **delta** value for a phase (e.g., **SrcCancellation**) during order creation. This makes it impossible to compute the start time, causing any instruction that relies on it to revert at runtime.

As a result, the taker is unable to cancel the escrow, effectively locking funds or disabling protocol logic.

This issue affects not just **cancel_escrow()**, but also:

- withdraw()
- public_withdraw()
- public_cancel_escrow()
- rescue_funds_for_escrow()

The attacker can selectively disable specific phases of the protocol, breaking intended behavior and fairness.

Remediation:

During order creation, validate that all computed phase start times (deployed_at + delta) fit within the u32 range, rejecting orders that would cause overflow.

This ensures critical protocol functions remain available and the system behaves as expected.

Commentary from the client:

"The backend will validate order parameters. Orders with malicious data will not be tracked or shared with resolvers."

OIN10-4 | Redundant Native Mint Check in Function cross-chain-escrow-src::cancel_order()

Fixed

Severity:	Probability:	Impact:
Informational	Very likely	Informational

Path:

programs/cross-chain-escrow-src/src/lib.rs#L385-L388

Description:

Within the function **cross-chain-escrow-src::cancel_order()**, it contains a check to verify the **mint** account's key matches with the variable **asset_is_native** of the provided order.

```
pub fn cancel_order(ctx: Context<CancelOrder>) -> Result<()> {
    let order = &ctx.accounts.order;

    require!(
        ctx.accounts.mint.key() == NATIVE_MINT || !order.asset_is_native,
        EscrowError::InconsistentNativeTrait
    );

    ...
}
```

However, this check is already indirectly enforced by a constraint defined in the **CancelOrder** context struct:

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

    #[account(
        constraint = mint.key() == order.token @ EscrowError::InvalidMint
    )]
    mint: Box<InterfaceAccount<'info, Mint>>,
    ...
}
```

The **mint.key()** must match the **order.token**. As established, the order.token is already validated against the Native Mint during order creation in the function **cross-chain-escrow-src::create()**:

```
pub fn create(
    ctx: Context<Create>,
    hashlock: [u8; 32], // Root of merkle tree if partially filled
    amount: u64,
    safety_deposit: u64,
    timelocks: [u64; 4],
    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,
    salt: u64,
    _dst_chain_params: DstChainParams,
) -> Result<()> {
    ...

    require!(
        ctx.accounts.mint.key() == NATIVE_MINT || !asset_is_native,
        EscrowError::InconsistentNativeTrait
    );

    ...
}
```

As a consequence, the native mint check within the **cancel_order()** function is redundant.

Remediation:

Consider removing the native mint check from the function **cross-chain-escrow-src::cancel_order()**.

OIN10-5 | Redundant signer Constraint in CancelOrderbyResolver Context

Fixed

Severity:	Probability:	Impact:
Informational	Very likely	Informational

Path:

programs/cross-chain-escrow-src/src/lib.rs#L987-L988

Description:

The **CancelOrderbyResolver** context struct declares the **resolver** account as **Signer<'info>**, which already enforces that the account must sign the transaction. However, the additional **signer** constraint in the attribute macro is redundant:

```
pub struct CancelOrderbyResolver<'info> {
    /// Account that cancels the escrow
    #[account(mut, signer)] /// [$audit-in4] signer is redundant here
    resolver: Signer<'info>,

    ...
}
```

Remediation:

It is recommended to remove the redundant **signer** constraint to improve code clarity and maintain consistency.

OIN10-6 | Missing validation of the mint account in the PublicCancelEscrow context

Fixed

Severity:	Probability:	Impact:
Informational	Rare	Informational

Path:

programs/cross-chain-escrow-src/src/lib.rs#L762
programs/cross-chain-escrow-src/src/lib.rs#L810
programs/cross-chain-escrow-src/src/lib.rs#L855
programs/cross-chain-escrow-src/src/lib.rs#L902
programs/cross-chain-escrow-dst/src/lib.rs#L297
programs/cross-chain-escrow-dst/src/lib.rs#L353
programs/cross-chain-escrow-dst/src/lib.rs#L395

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>>,
    ...
}
```

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.

The issue applies to the following functions:

- cross-chain-escrow-src::withdraw()
- cross-chain-escrow-src::public_withdraw()
- cross-chain-escrow-src::cancel_escrow()
- cross-chain-escrow-src::public_cancel_escrow()
- cross-chain-escrow-dst::withdraw()
- cross-chain-escrow-dst::public_withdraw()
- cross-chain-escrow-dst::cancel()

Remediation:

Consider adding a constraint for the account **mint** to make sure it matches with the **escrow.token**.

```
++  #[account(constraint = mint.key() == order.token @
EscrowError::InvalidMint)]
    mint: Box<InterfaceAccount<'info, Mint>>,
```

OIN10-7 | Redundant Instruction Parameters

Fixed

Severity:	Probability:	Impact:
Informational	Very likely	Informational

Path:

programs/cross-chain-escrow-dst/src/lib.rs#L236-L237
programs/cross-chain-escrow-src/src/lib.rs#L679-L680

Description:

In **cross-chain-escrow-dst::Create<>**, the **safety_deposit** parameter in the instruction is redundant as it is not used in any constraint or account derivation logic.

Similarly, in **cross-chain-escrow-src::CreateEscrow<>**, the **dutch_auction_data** parameter is redundant in the instruction.

```
#[instruction(order_hash: [u8; 32], hashlock: [u8; 32], amount: u64,
safety_deposit: u64, recipient: Pubkey)]
pub struct Create<'info> {
```

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

Remediation:

Remove these unused instruction parameters to keep the interface clean and avoid confusion.


hexens
x
1inch