OpenZeppelin | security

Fusion Swap for Solana Audit

1inch
NETWORK

April 29, 2025

Table of Contents

Summary

Type	Defi
Timeline	From 2025-03-31 To 2025-04-08
Languages	Rust

Total Issues	11 (6 resolved, 1 partially resolved)
Critical Severity Issues	0 (0 resolved)
High Severity Issues	0 (0 resolved)
Medium Severity Issues	1 (0 resolved)
Low Severity Issues	2 (0 resolved, 1 partially resolved)
Notes & Additional Information	8 (6 resolved)

Scope

OpenZeppelin audited the 1inch/solana-fusion-protocol repository at commit 8743d38.

In scope were the following files:

```
programs
├── fusion-swap
│   └── src
│       ├── auction.rs
│       ├── error.rs
│       └── lib.rs
└── whitelist
    └── src
        ├── error.rs
        └── lib.rs
```

System Overview

1inch Fusion is a decentralized exchange that facilitates token swaps while being resistant to front-running, leveraging a Dutch auction mechanism. Instead of having a public, open order book, the system utilizes an escrow-based approach where users (`makers`) create specific swap orders. These orders define the source asset and amount to be sold, along with the minimum amount of the destination asset that they are willing to receive.

The system employs a time-decaying exchange rate, being more favorable to the maker at the beginning and then gradually decreasing towards their minimum acceptable rate over the order's duration. A network of whitelisted actors (`resolvers` or `takers`) monitor these orders and compete to fill them when the rate becomes profitable according to their strategies. This design aims to protect users from MEV activities like sandwich attacks commonly encountered in public AMM pools.

The system comprises two main Solana programs:

1. **`fusion_swap`** (`5uzpYuGqBaetRMXPDtGWGN9W4mdmgBzpGHcQACrZ1npi`): Handles the core logic of order creation, escrow management, order filling, and cancellation.
2. **`whitelist`** (`DyXFcRxGWFoMz1j76SeMXHjQqZKudLXeJY3h1K7BNJiQ`): Manages the set of authorized addresses (resolvers) permitted to fill or cancel expired orders within the `fusion_swap` program.

Order lifetime

The lifecycle of a swap order involves several potential paths.

1. **Creation:** A user (`maker`) initiates the process by calling the `create` instruction. They provide the `OrderConfig`, which includes:
    - Source token and amount (`src_amount`)
    - Destination token and the minimum acceptable amount (`min_dst_amount`)
    - An estimated destination amount (`estimated_dst_amount`), used for surplus fee calculation
    - Order expiration timestamp (`expiration_time`)
    - Details for the Dutch auction (`dutch_auction_data`).

    - Fee configuration (`fee`), including potential protocol, integrator, and cancellation fees
    - Flags indicating whether source/destination assets are native SOL (`src_asset_is_native`, `dst_asset_is_native`)

The `maker` signs the transaction, and their specified `src_amount` in SPL tokens or SOL is sent to a dedicated escrow token account (an Associated Token Account - ATA). In case of SOL it is wrapped into wSOL. The ATA is owned by a Program Derived Address (PDA) unique to the order details (`escrow` account), with the PDA as its authority. Various checks ensure order validity (e.g., non-zero amounts, valid expiration, consistent fee configs).

1. **Filling:** A whitelisted `taker` (resolver) finds a suitable order and calls the `fill` instruction.
    - The taker specifies the order details (`OrderConfig`) and the `amount` of the source token they wish to purchase (allowing partial fills).
    - The system verifies that the order has not expired and that the taker is whitelisted (possesses a valid `ResolverAccess` account from the `whitelist` program).
    - It calculates the required `dst_amount` the taker must pay, factoring in the Dutch auction rate adjustment based on the current time using `calculate_rate_bump`.
    - The specified `amount` of source tokens is transferred from the `escrow_src_ata` to the `taker_src_ata`.
    - The calculated `dst_amount` is paid by the taker. This amount is then divided as follows:
        - Integrator fee goes to the `integrator_dst_acc` (if applicable).
        - Protocol fee (including any surplus fee) goes to the `protocol_dst_acc` (if applicable).
        - The remaining amount goes to the `maker_receiver` (or their `maker_dst_ata`).
    - If the fill completes the entire `src_amount`, the `escrow_src_ata` is closed, and its lamport balance (rent) is returned to the `maker`.
2. **Cancellation by Maker:** The original `maker` can decide to cancel their order by calling the `cancel` instruction.
    - The maker provides the unique `order_hash` derived from the order parameters.
    - The instruction verifies that the `maker` is the signer.
    - Any remaining source tokens in `escrow_src_ata` are transferred back to `maker_src_ata`.

    - `escrow_src_ata` is closed, returning its lamport balance (rent) to the `maker`.

3. **Cancellation By Resolver:** If an order expires, it becomes eligible for cancellation by a whitelisted resolver via the `cancel_by_resolver` instruction. This mechanism incentivizes cleaning up expired orders.

    - A whitelisted `resolver` calls the instruction, providing the `OrderConfig` and a `reward_limit` they are willing to accept.
    - The system verifies that the order is expired and that the caller is whitelisted.
    - The system checks if cancellation by resolvers is permitted.
    - If the source asset was *not* native SOL, the remaining tokens are transferred back to the `maker_src_ata`.
    - A `cancellation_premium` is calculated based on the time elapsed since expiration (`calculate_premium`), capped by `order.fee.max_cancellation_premium`.
    - The `escrow_src_ata` is closed. Crucially, its *entire* lamport balance (rent + any principal if the source was wrapped SOL) is transferred to the `resolver`.
    - The `resolver` then immediately transfers a portion of these received lamports back to the `maker`. The amount returned to the maker is the total received minus the calculated `cancellation_premium` (further capped by the `reward_limit` provided by the resolver). The resolver keeps the premium as a reward.

Dutch Auction Implementation

The Dutch auction modifies the exchange rate over time, making the order progressively cheaper for the taker. It is implemented via the `AuctionData` struct within the `OrderConfig` and `calculate_rate_bump` functions.

- **Configuration:** `AuctionData` contains:
    - `start_time`: The Unix timestamp when the auction begins.
    - `duration`: The total length of the auction period.
    - `initial_rate_bump`: A starting adjustment (in basis points, `BASE_1E5`) applied to the destination amount. A positive bump means the taker initially pays *more* than the base rate derived from `min_dst_amount`.
    - `points_and_time_deltas`: A vector defining points on a curve. Each point specifies a `rate_bump` and the `time_delta`. This allows for defining custom decay curves.

- **Calculation:** The `calculate_rate_bump` function determines the applicable rate bump based on the current `timestamp`:
    - Before `start_time`, the `initial_rate_bump` is used.
    - After `start_time + duration`, the rate bump is 0 (meaning the rate reverts to the one implied by `min_dst_amount`).
    - During the auction, the function iterates through the `points_and_time_deltas`. It finds the segment of the curve corresponding to the current `timestamp` and performs linear interpolation between the `rate_bump` values of the segment's start and end points to find the current bump.

Fees

The protocol incorporates several types of fees, configured within the `OrderConfig.fee` (`FeeConfig` struct):

1. **Protocol Fee:** A percentage (`protocol_fee`, basis points relative to `BASE_1E5`) of the total `dst_amount` paid by the taker during a `fill`. This fee is directed to the `protocol_dst_acc` if provided.
2. **Integrator Fee:** A percentage (`integrator_fee`, basis points relative to `BASE_1E5`) of the total `dst_amount` paid by the taker during a `fill`. This fee is directed to the `integrator_dst_acc` if provided, allowing UI providers or integrators to earn revenue.
3. **Surplus Fee (Positive Slippage):** If the actual amount the maker receives (after the protocol and integrator fees are deducted from `dst_amount`) exceeds the `estimated_dst_amount` provided in the order, a percentage (`surplus_percentage`, basis points relative to `BASE_1E2` = 100%) of this positive difference (surplus) is taken as an additional fee. This surplus fee is added to the protocol fee amount.
4. **Cancellation Premium:** Configured via `max_cancellation_premium` (an absolute lamport amount). When a resolver cancels an expired order using `cancel_by_resolver`, they earn a premium calculated based on the time elapsed since expiration, capped by this value. This fee is effectively paid by the maker from the funds held in the escrow ATA (specifically its lamport balance).

Whitelist

The `whitelist` program serves as an access control layer for specific actions within the `fusion_swap` program.

- **Purpose:** It restricts the ability to act as a `taker` (calling `fill`) and to cancel expired orders (calling `cancel_by_resolver`) to only authorized addresses. This fulfills the "network of professional market makers" concept.
- **Mechanism:**
  - The program maintains a central `WhitelistState` account (a PDA seeded by `WHITELIST_STATE_SEED`) which stores the `owner` public key.
  - The `owner` has the authority to manage the whitelist.
  - To whitelist a user (resolver), the owner calls `register`, providing the user's public key. This creates an empty `ResolverAccess` account (a PDA seeded by `RESOLVER_ACCESS_SEED` and the user's key). The existence of this account signifies that the user is whitelisted.
  - To remove a user, the owner calls `deregister`, which closes the user's `ResolverAccess` account.
  - The `fill` and `cancel_by_resolver` instructions in `fusion_swap` include constraints that verify that the transaction signer (`taker` or `resolver`) has a valid `ResolverAccess` account derived using the `whitelist` program's ID and the correct seeds.

Security Model and Trust Assumptions

Users and participants in this protocol operate under several security assumptions and known risks:

- **Whitelist Reliance:** The security and liveness of the filling process depend entirely on the whitelisted resolvers. Users trust the `whitelist` program's owner to:
  - Only whitelist competent and non-malicious resolvers.
  - Maintain a sufficient set of active resolvers to ensure that orders are filled.
  - Not arbitrarily remove resolvers or transfer ownership to untrusted parties.

- **Resolver Behavior:** Users trust that whitelisted resolvers will act economically rationally and fill orders when profitable. There is no on-chain guarantee that an order will be filled, even if the rate becomes favorable.

- **Off-Chain Dependencies:** Order discovery and potentially submission likely rely on off-chain infrastructure (e.g., APIs, front-ends like 1inch's). Users and resolvers trust this infrastructure to be available, accurate, and censorship-resistant. Downtime or manipulation of this layer can prevent order creation or filling. The centralized backend is critical for order flow. If it becomes unavailable or behaves incorrectly, valid orders might not be forwarded to takers, leading to degraded protocol usability.

  Conversely, if the backend fails to correctly filter invalid orders, they may still be passed along, undermining the integrity of the off-chain matching process. The backend introduces a layer of trust that contradicts the trust-minimized ethos of decentralized systems. Users and takers must assume that the backend is not censoring or selectively delaying orders. Although the on-chain program uses PDA checks to ensure the integrity of orders at fill time, it cannot prevent the backend from influencing which orders are seen or prioritized.

- **Expiration Handling:** Expired orders rely on resolvers calling `cancel_by_resolver` for cleanup, incentivized by the cancellation premium. If no resolver cancels, the funds (especially native SOL) remain in the escrow ATA until manually cancelled by the maker or eventually by a resolver. Users trust that this incentive mechanism is sufficient.

Privileged Roles

Several roles possess special capabilities within the system:

1. **Whitelist Owner:**
    - **Description:** The single address designated as the owner in the `whitelist` program's `WhitelistState` account.
    - **Capabilities:**
        - Add resolvers (`register`)
        - Remove resolvers (`deregister`)
        - Transfer ownership of the whitelist (`transfer_ownership`)

2. **Resolvers / Takers:**

    - **Description:** Addresses that have been whitelisted by the Whitelist Owner via the `register` function, resulting in the creation of a corresponding `ResolverAccess` PDA.
    - **Capabilities (within `fusion_swap`):**
        - Fill active orders (`fill`)
        - Cancel expired orders for a premium (`cancel_by_resolver`)

Medium Severity

M-01 Lack of Event Emission

None of the instructions in either program emit events upon execution. This omission hinders transparency and external observability. Without emitted events, off-chain consumers such as dashboards, indexers, and other smart contracts must rely on custom transaction-parsing logic to infer state changes like order creation, fulfillment, or cancellation. This increases implementation complexity and creates a brittle dependency on internal instruction formats.

Although transaction data is publicly available on-chain, the lack of standardized event emissions significantly reduces the ease of monitoring protocol activity. This design choice can impact the developer and user experience, as protocols that emit events allow for more accessible and standardized tracking of meaningful on-chain events.

Consider emitting events after sensitive changes take place to facilitate tracking and notify off-chain clients that are following the programs' activity.

Update: *Acknowledged, not resolved. The 1inch team stated:*

> *Adding events means increasing the transaction's CU which we cannot yet justify.*

Low Severity

L-01 Lack of External Documentation

The `solana-fusion-protocol` repository is missing fundamental project information, including a `README.md` file, project description, and any form of documentation directory or usage guides. This significantly reduces the accessibility and maintainability of the codebase, especially for new contributors, auditors, and external developers attempting to understand or integrate with the protocol. The absence of a `README.md` also means that there is neither clear guidance on how to set up, test, or deploy the project, nor any details on its purpose, architecture, or dependencies.

Including a basic `README.md` with setup instructions, usage examples, and an overview of the protocol is a widely accepted best practice in open-source and production codebases. This ensures that the project can be reliably used and reviewed, and it also helps establish the credibility and usability of the code.

Consider at least adding the following to the documentation:

- A short description of the protocol
- Setup and build instructions
- How to run tests
- Contract architecture and module descriptions

Update: *Partially resolved in pull request #72. The The 1inch team stated:*

> *Noted. We added a whitepaper and we will add a basic README.md in future versions.*

L-02 Missing Docstrings

Key parts of the `fusion_swap` program lack essential docstrings, reducing clarity and increasing the risk of misuse:

- **Program Module** (**`#[program]`**): Missing top-level docstring explaining the contract's purpose and functionality.
- **Instruction Handlers** (**`create`**, **`fill`**, **`cancel`**, **`cancel_by_resolver`**): No documentation on purpose, parameters, expected preconditions, side effects, or error cases.
- **`OrderConfig`** **Struct**: Lacks docstring for the struct and its fields (`id`, `src_amount`, `min_dst_amount`, `expiration_time`, etc.), which are central to order logic.
- **`UniTransferParams`** **Enum**: No explanations for the enum or its variants (`NativeTransfer`, `TokenTransfer`), which abstract token transfers.
- **Helper Functions**: `order_hash`, `get_fee_amounts`, and `uni_transfer` lack docstrings describing logic, parameters, and expected behavior.

Consider adding concise documentation to the aforementioned areas. Doing this would greatly improve maintainability, readability, and safety for users and auditors.

Update: *Acknowledged, will resolve. The 1inch team stated:*

> *Noted. We will add the essential documentation in future versions.*

Notes & Additional Information

N-01 `transfer_ownership` Performs Immediate Ownership Transfer Without Safeguards

The `transfer_ownership` function in the whitelist program assigns ownership to the `_new_owner` address immediately upon invocation. This approach introduces risk, as an incorrect or unintended address may be set as the new owner due to human error.

Without a confirmation mechanism, such as a two-step ownership transfer (e.g., `proposeOwner` and `acceptOwnership`), there is no opportunity to recover from a misconfiguration. If ownership is accidentally assigned to an unwanted contract address, a burn address, or an address not controlled by the intended recipient, the contract may become irreversibly inaccessible or mismanaged.

To mitigate this, consider implementing a two-step transfer pattern, where the new owner must explicitly accept ownership before the change is finalized. Alternatively, if there are guarantees in the system design that ensure safe use of the current one-step transfer mechanism, they should be clearly documented to justify the approach.

Update: *Acknowledged, not resolved. The 1inch team stated:*

> *In the unlikely event control over the whitelist contract is lost, redeployment to a new address would suffice without significantly disrupting protocol functionality. Hence, we do not see a strong need for additional checks.*

N-02 Ambiguous Use of "Owner" in Whitelist Program May Cause Confusion

In the context of the `whitelist` program, the term "owner" is used to refer to the actor who controls the whitelist state. However, this terminology can be misleading within the Solana ecosystem, as in this blockchain network, an account's `owner` is the program ID that has permission to modify the account's data. This is distinct from any authority or admin-like key that might control the behavior or state within a program.

This ambiguity can be particularly problematic for developers familiar with Ethereum, where "owner" often connotes a privileged user role rather than program ownership.

To improve clarity and maintain consistency with Solana conventions, consider using more precise terminology such as `authority`, `admin`, or `controller`.

Update: *Resolved in [pull request #84](#).*

N-03 Misleading Underscore Prefix on Used Argument `_new_owner`

The `transfer_ownership` function takes `_new_owner` as a parameter, which is used within the function body. However, the underscore prefix conventionally signals that a parameter is intentionally unused. This creates a misleading impression that `_new_owner` is not used, potentially confusing readers or maintainers.

To improve code clarity and adhere to conventional naming practices, consider removing the underscore from `_new_owner`.

Update: *Resolved in [pull request #83](#).*

N-04 Redundant `.key()` Call on a `Pubkey` Value

In the `transfer_ownership` function, the `whitelist_state.owner = _new_owner.key();` assignment is redundant because `_new_owner` is already a `Pubkey`. The `.key()` method is typically used on `AccountInfo` objects to retrieve the `Pubkey`. However, in this case, calling `.key()` on a `Pubkey` simply returns itself, adding unnecessary verbosity and potentially confusing readers.

Consider replacing the `whitelist_state.owner = _new_owner.key();` assignment with `whitelist_state.owner = _new_owner;` to make the code more concise and idiomatic.

Update: *Resolved in [pull request #82](#).*

N-05 Programs Use Outdated Anchor Dependencies

Currently, the programs rely on outdated versions of both the `anchor-lang` and `anchor-spl` crates. Since their release, there is a new version containing bug fixes, improved developer ergonomics, and new features that may enhance the safety and maintainability of the codebase.

Using outdated dependencies can expose the program to known vulnerabilities or bugs already addressed in newer versions. It may also hinder the adoption of best practices and reduce compatibility with other up-to-date tooling in the ecosystem.

Consider upgrading to the latest versions of the `anchor-lang` and `anchor-spl` crates, ensuring that the changes introduced in the newer versions are compatible with the current codebase.

Update: *Resolved in pull request #80.*

N-06 The `toolchain` Section in `Anchor.toml` Is Empty

The `Anchor.toml` file is missing a specified `toolchain` version. Omitting this can lead to inconsistencies between the Anchor CLI version used by different developers/auditors and the one expected by the project. Version mismatches may introduce subtle bugs, compilation errors, or unexpected behavior, especially if breaking changes have been introduced in newer releases.

Defining the toolchain version helps ensure reproducibility of builds and a consistent development environment across teams and CI systems. It also improves clarity for auditors reviewing the code under a known Anchor version.

To mitigate this, specify the expected Anchor CLI version in the `toolchain` section of `Anchor.toml`.

Update: *Resolved in pull request #81.*

N-07 Program File Contains Too Many Lines of Code

The `fusion_swap` program currently implements all instructions within a single, large file. This monolithic structure negatively impacts readability, collaboration, and future scalability.

Smaller, instruction-specific modules are easier to understand and navigate. When each instruction (e.g., `create` and `cancel`) and its associated components (such as its `Accounts` struct) are located in separate files, developers can more easily comprehend and maintain the codebase. This modularization also reduces the likelihood of merge conflicts, particularly when multiple team members are working on different instructions concurrently. Moreover, as the program expands in size or complexity, the current flat structure may become increasingly difficult to manage, making debugging or refactoring more error-prone.

Consider splitting the `fusion_swap` program by putting each instruction into a separate module or file. This would help bring clarity to the codebase, facilitate team collaboration, and improve long-term maintainability.

Update: *Acknowledged, not resolved. The 1inch team stated:*

> *Noted — we've decided not to proceed with changes at this time.*

N-08 Incorrect Docstrings

Throughout the codebase, multiple instances of docstrings containing technically incorrect or misleading information were identified:

1. `/// Account to store order conditions` (Present in lines 415, 501,577, and 638 in `fusion-swap/src/lib.rs`). This description inaccurately suggests that the escrow account stores order conditions. In reality, it is a PDA used as the authority for the escrow source token account. Its address is derived from order details (`order_hash`), but it does not store the order configuration directly. A clearer description would be: `/// PDA derived from order details, acting as the authority for the escrow ATA`
2. `/// size(timestamp) + size(rate_bump) < 64` (Present on lines 37 and 50 on `auction.rs`). This statement is factually incorrect. `timestamp` is a `u64` (64 bits), and `rate_bump`, though originally a `u16` value, is cast to `u64` for arithmetic purposes. Even considering the original type, the combined bit size is `64 + 16 = 80`, which is

not less than 64. The intention appears to be to justify that the `time_difference * rate_bump` multiplication cannot overflow a `u64` container. In practice, the values are constrained:

- Time differences are derived from a `u16` (maximum 65535) value.
- `rate_bump` values also originate from a `u16` value.

Therefore, the maximum multiplication result is approximately `65535 * 65535 ≈ 2^32`, which fits safely within a `u64` container. While the logic is sound, the justification based on bit sizes is flawed and should be clarified.

Clear and accurate docstrings are essential for correctness, maintainability, and auditability. As such, consider addressing the aforementioned instances of incorrect/misleading docstirngs.

Update: *Resolved in pull request #85.*

Conclusion

1inch Fusion is a decentralized exchange on the Solana blockchain that facilitates token swaps that are resistant to front-running, leveraging a Dutch auction mechanism. Instead of using a public order book, users create escrowed swap orders with defined minimum returns. The exchange rate decays over time (based on a Dutch auction model), starting favorably for the maker and dropping until a whitelisted actor accepts the trade. This setup helps prevent MEV attacks.

The implementation reflects a solid understanding of Solana development, with robust handling of edge cases, a comprehensive test suite, and thoughtful design choices. While no critical vulnerabilities were found, some minor issues were identified and actionable recommendations were provided to improve code maintainability, adherence to best practices, documentation, and overall clarity. The 1inch team is appreciated for being highly responsive and collaborative throughout the process.