Zellic

Drift Protocol

DeFi Security Assessment

February 16, 2022

Prepared for:

Cindy Leow and Chris Heaney

Drift Protocol

Prepared by:

Stephen Tong and Jasraj Bedi

Zellic Inc.

Contents

About Zellic

Zellic was founded in 2020 by a team of blockchain specialists with more than a decade of combined industry experience. We are leading experts in smart contracts and Web3 development, cryptography, web security, and reverse engineering. Before Zellic, we founded perfect blue, the top competitive hacking (CTF) team in the world. Since then, our team has won countless cybersecurity contests and blockchain security events.

Zellic aims to treat clients on a case-by-case basis and to consider their individual, unique concerns and business needs. Our goal is to see the long-term success of our partners rather than to simply provide a list of present security issues. Similarly, we strive to adapt to our partners' timelines and to be as available as possible.

To keep up with our latest endeavors and research, check out our website https://zellic.io and follow @zellic_io on Twitter. If you are interested in partnering with us, please email us at hello@zellic.io or contact us on Telegram at https://t.me/zellic_io.

1. Introduction

1.1. About Drift

Drift is a lightning-fast perpetual futures DEX on Solana with on-chain, cross-margined perpetual futures. Drift's goal is to bring a state-of-the-art trader-centric experience from centralized exchanges on-chain.

More concretely, Drift is a trading platform where users can take on leveraged positions with perpetual swaps on various token pairs. Users open and close these positions by trading against a virtual AMM for each token pair. For example, a user may open a 5x long Solana position by buying SOL-PERPs on the AMM, collateralized by a USDC deposit. Users with underwater positions are liquidated by keepers, who are rewarded with a liquidation fee. On a devnet beta deployment, users can also place limit and stop orders. These orders are also filled by keepers in a similar fashion.

1.2. Methodology

During a security assessment, Zellic works through standard phases of security auditing including both automated testing and manual review. These processes can vary significantly per engagement, but the majority of the time is spent on a thorough manual review of the entire scope.

Alongside a variety of open-source tools and analyzers used on an as-needed basis, Zellic focuses primarily on the following classes of security and reliability issues:

Basic coding mistakes. Many critical vulnerabilities in the past have been caused by simple, surface-level mistakes that could have easily been caught ahead of time by code review. We analyze the scoped smart contract code using automated tools to quickly sieve out and catch these "shallow" bugs. Depending on the engagement, we may also employ sophisticated analyzers such as model checkers, theorem provers, fuzzers, etc. as necessary. We also perform a cursory review of the code to familiarize ourselves with the contracts. For this assessment, we additionally look for Solana-specific mistakes such as account forgery, type confusion vulnerabilities, and transaction compute quota exhaustion.

Business logic errors. Business logic is the heart of any smart contract application. We manually review the contract logic to ensure that the code implements the expected functionality as specified in the platform's design documents. We also thoroughly examine the specifications and designs themselves for inconsistencies, flaws, and vulnerabilities. This involves use-cases that open the opportunity for abuse, such as flawed share pricing, unintended arbitrage opportunities, infinite leverage bugs, etc.

Complex integration risks. Several high-profile exploits have been the result of not any bug within the contract itself, but rather an unintended consequence of its interaction with the broader DeFi ecosystem. We perform a meticulous review of all of the contract's possible external interactions, and summarize the associated risks; for example: oracle price manipulation, economic / market microstructure manipulation (market breaking), game-theoretic attacks, potential incentivization and market failures, flash loan attacks, etc.

Code maturity. We review for possible improvements in the codebase in general. We look for violations of industry best practices and guidelines, or code quality standards. We also provide suggestions for possible optimizations, such as fee & rent optimization, upgradeability weaknesses, centralization risks, etc.

For each finding, Zellic assigns it an *impact* rating based on its *severity* and *likelihood*. There is no hard-and-fast formula for calculating a finding's impact; we assign it on a case-by-case basis based on our professional judgment and experience. As one would expect, both the severity and likelihood of an issue affect its impact; for instance, a highly severe issue's impact may be attenuated by a very low likelihood. We assign the following impact ratings (ordered by importance): *Critical*, *High*, *Medium*, *Low*, and *Informational*.

Similarly, Zellic organizes its reports such that the most *important* findings come first in the document, rather than impact alone. Thus, we may sometimes emphasize a "Informational" finding higher than a "Low" finding. The key distinction is that although certain findings may have the same *impact rating*, their *importance* may differ. This varies based on numerous soft factors, such as our clients' threat model, their business needs, project timelines, etc. We aim to provide useful and actionable advice to our clients that consider their long-term goals, rather than simply a list of security issues at present.

1.3. Scope

The engagement involved a review of the following targets:

Protocol-V1

Repository	https://github.com/drift-labs/protocol-v1
Versions	`70e9ef1f079a69bcc158fa8dbad6507654ec75b6` `(zellic-audit branch)` `f0016ce758a8b6ee1b01f43eeac4208d1505d61f` `(limit-orders branch)`
Language	Rust
Platform	Solana

1.4. Disclaimer

This assessment does not provide any warranties on finding all possible issues within its scope; i.e., the evaluation results do not guarantee the absence of any subsequent issues. We, of course, also cannot make guarantees on any additional code added to the assessed project after our assessment has concluded. Furthermore, because a single assessment can never be considered comprehensive, we always recommend multiple independent assessments paired with a bug bounty program. Finally, this assessment report should not be considered as financial or investment advice.

2. Executive Summary

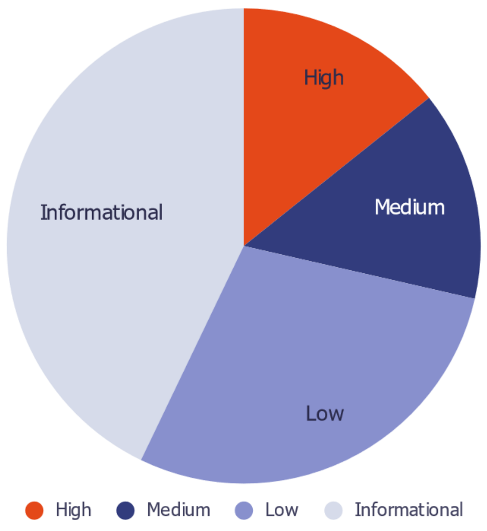
We audited the scoped contracts and we discovered 7 findings. One finding was of high importance: an arbitrage scenario where an attacker can exploit other market participants. Of the remaining findings, 1 was of medium impact, and 2 were of low impact. The remaining findings were informational in nature. Overall, we applaud Drift for their attention to detail and diligence in maintaining high code quality standards. The code base is clean, clear, and well-maintained.

Drift is a sophisticated trading platform with several complex features. A serious bug in the business logic of these features would be critical. Therefore, for this audit, we focused heavily on potential economical attacks for this audit in addition to typical Solana coding mistakes.

In addition to a careful manual review of the codebase, we also developed a suite of basic fuzz tests to further ensure the correctness of the platform. These tests essentially send random instructions to the program, while checking critical business logic invariants, such as margin requirements. We discuss this further in Appendix A.

Breakdown of Finding Impacts

Impact Level	Count
Critical	0
High	1
Medium	1
Low	2
Informational	3



3. Detailed Findings

3.1. Possible sandwich attack on stop loss/trigger orders

- **Target:** ClearingHouse
- **Severity:** High
- **Impact:** High
- **Category:** Business Logic
- **Likelihood:** High

Description

Users may place trigger market and trigger limit orders, which can only be filled when the current mark price satisfies the order's trigger condition. For example, a user may place a stop loss by placing market sell which triggers on the mark price falling below a specified stop price. Conversely, a user may also place an order to buy swaps if the mark price exceeds a threshold.

In theory, a user's stop loss (or buy stop) should not be filled unless the true price of the asset has fallen below the user-specified threshold. However, an attacker with access to a large amount of liquidity can temporarily manipulate the AMM past the trigger price then forcibly fill the user's order. This is possible because Drift validates trigger conditions using the current AMM mark price, without TWAP or Oracle validation.

Crucially, if the victim trigger order trades in the same direction as the trigger condition (i.e., above+buy, below+sell), this attack can be profitable for the attacker. After manipulating the market, the attacker can force-fill any users who have similar orders open. Depending on the market conditions, this could present a highly profitable arbitrage opportunity for the attacker, while ultimately being detrimental to other market participants.

The attack is limited to ±10% of the current oracle price, as Drift rejects transactions that push the oracle-mark spread beyond this interval. Despite these limitations, in our experiments and calculations, we discovered that this strategy can still be lucrative for an attacker.

Attack Scenario

In the following discussion, we consider a buy-stop order which longs (i.e., buys) swaps when the mark price exceeds 5% more than the current price. The scenario for stop-loss orders are inverse to this scenario and effectively the same.

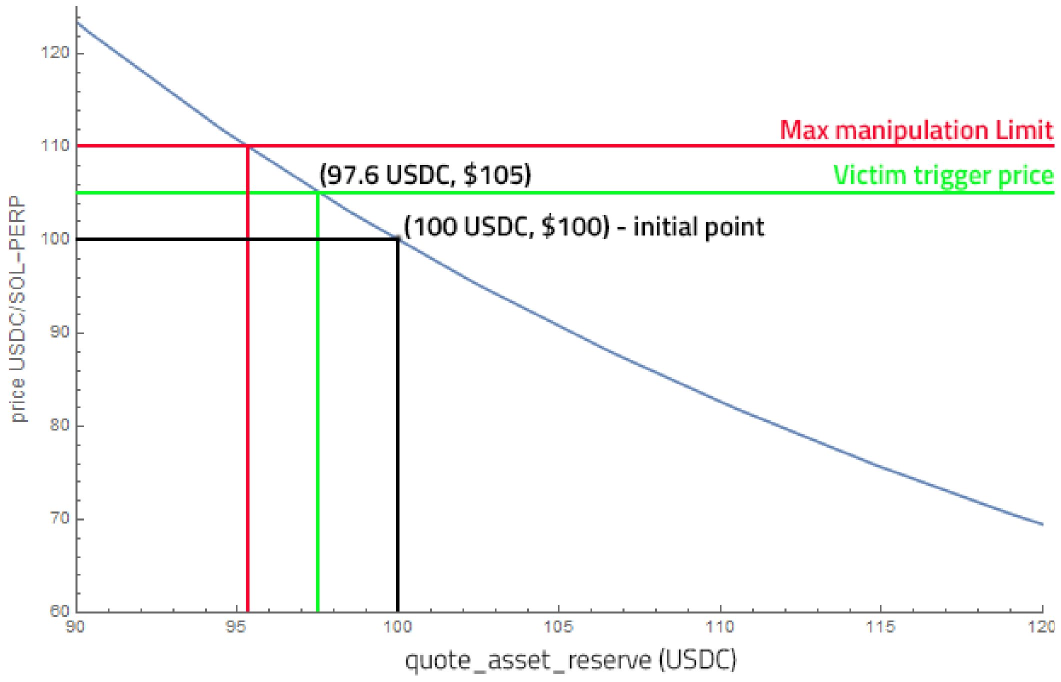
Suppose the SOL-PERP/USDC market's virtual AMM has 100 SOL-PERP and 100 USDC in reserve, and the peg multiplier is 100. The initial state of the AMM is as follows:

base_asset_reserve	quote_asset_reserve	invariant_k	peg_multiplier	price
100 (SOL-PERP)	100 (USDC)	100000	100.0	100 USDC / SOL-P.

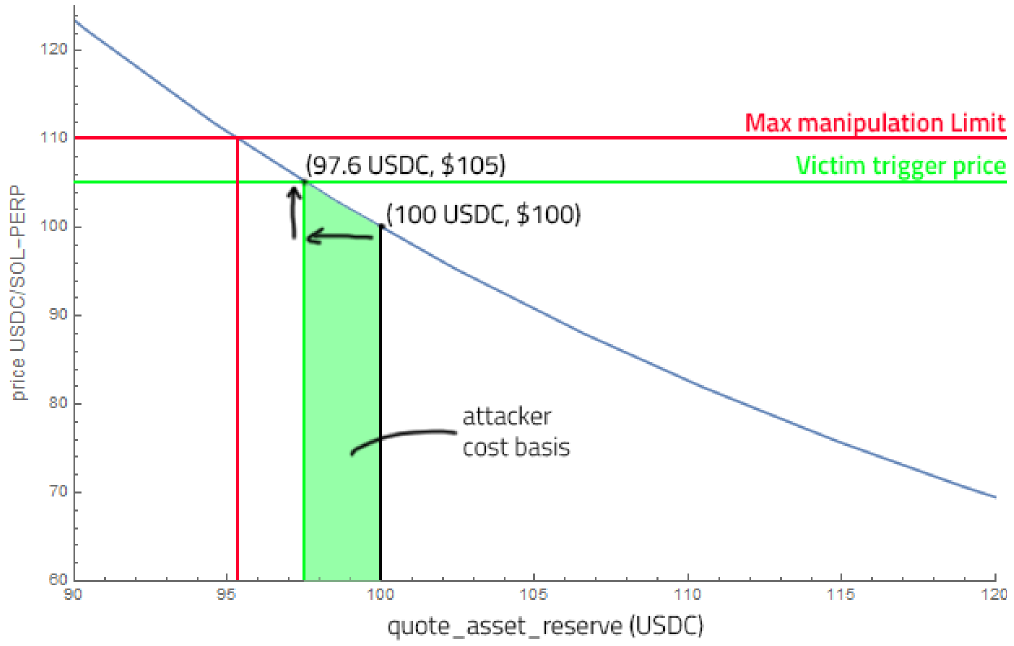
Suppose a victim has an open order, as described above:

Order Type	Direction	Trigger Condition	Trigger Price	Limit Price	Volume
Trigger Limit	LONG	ABOVE	$105	$110	10000 SOL-PERP

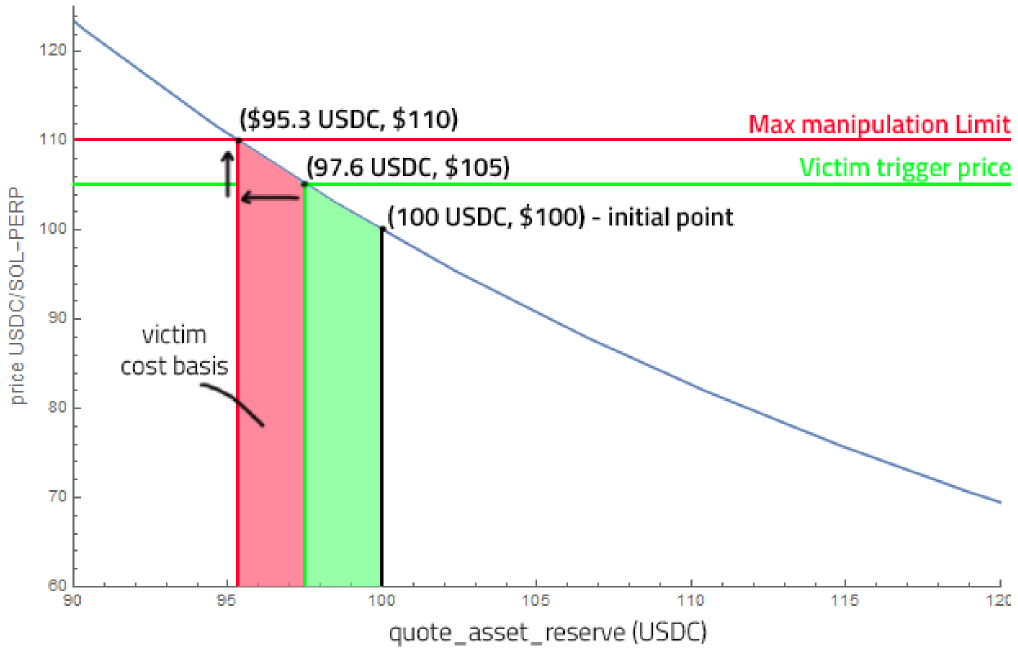
The following graph shows the current market state:



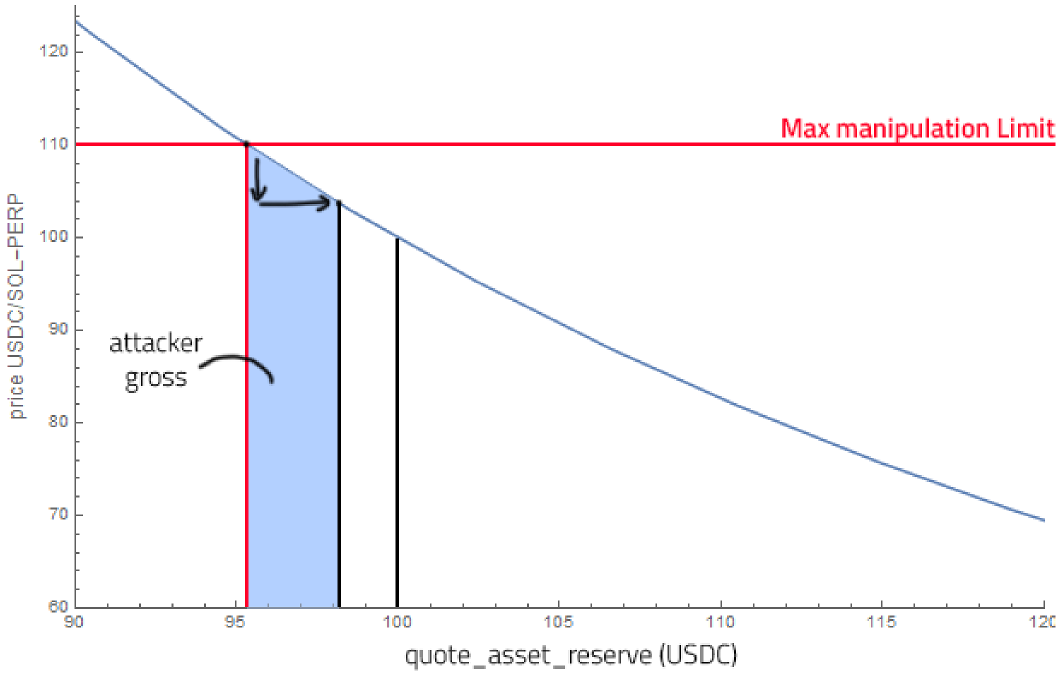
The attacker will then open a large LONG position on the market to push the AMM up the curve past the victim's trigger price:



Now that the AMM has moved past the victim's trigger price, the attacker can now trigger a fill of the victim order. This pushes the AMM further up the curve:



The attacker now dumps the USDC they acquired at a lower average cost than the now inflated price, for a profit:



This buy-fill-sell "sandwich" allows the attacker to sell USDC at a higher price than the average cost, generating profits for the attacker. These profits ultimately come from force-filling the victim at an unfavorable price above the market. Inversely, an attacker can perform a sell-fill-buy sandwich to attack users with open stop loss orders. In such a scenario, the victim essentially is made to sell swaps at below market price.

The attacker must pay a small fee (0.1% taker fee both ways) to conduct the market manipulation, but we find that in many scenarios, the attack remains profitable nonetheless.

Impact

We developed a Proof-of-Concept (PoC) attack and tested it on our local fork of Drift's devnet deployment. We show that the market is profitable on real deployments of the Drift platform:

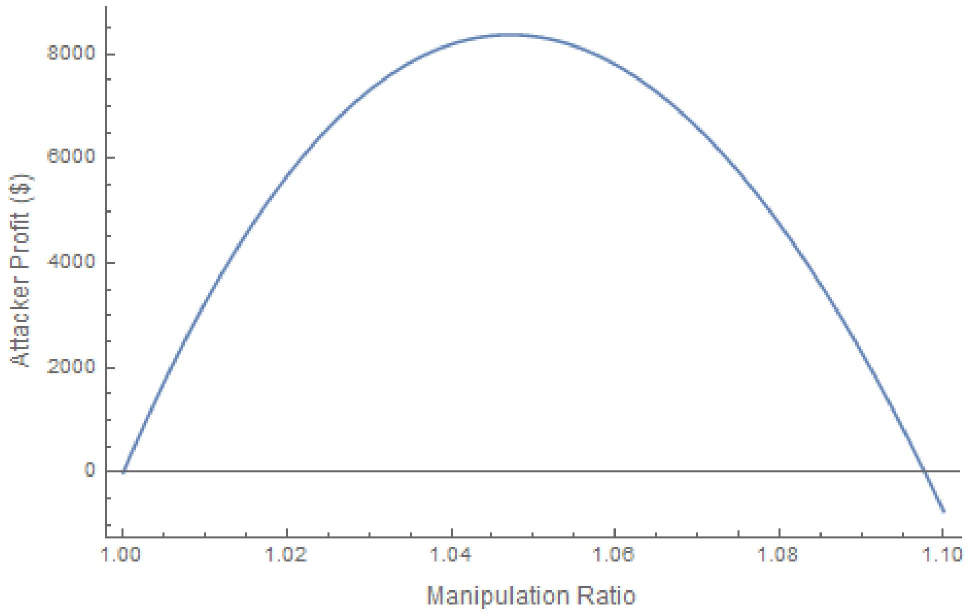
```
Cloning devnet state...
  damm = 3PfbDmWxR6e2rJ2brhSv7KJyUrHbSCu63d3FHqdLhxUJ
  order_state = J1ZQ3HMrGrsk9k2vVcYi6Wgd1kSVHPULtR5JaKjXhZXm
  order_history = AnyKcrUGyz5mywgdiREx5q5HawL6U6Yuenv2owhzBLo6
  Loading 3PfbDmWxR6e2rJ2brhSv7KJyUrHbSCu63d3FHqdLhxUJ from cluster
  Loading 25oHbe7vNbhUe43z4TkEYRcKYgewT2TaJUyrD2c9msV3 from cluster
  ...
  Loading J83w4HKfqxwcq3BEMMkPFSppX3gqekLyLJBexebFVkix from cluster
Done! Running PoC.

Initial state:
 - Victim starting collateral: 1000000.000000
 - Hacker starting collateral: 1000000.000000
 - Starting mark price: 92.9201093497

* Place victim order
 - OK, order placed (buy stop @ 97.566)

* Move market up
Initial AMM state:
 - USDC reserve = 67505.5958253027788
 - SOL  reserve = 79390.1778400130637
 - K = 5359281257766831058356052075349l7225
 - Peg = 109.279x

Calculate order size for market manipulation:
 - New USDC reserve = 69172.6516848645376
 - Order size: 182174197277 ($182174)

* Moved market up
 - New mark price: 97.5661148172

* Fill victim buy order
 - New sol price: 102.2121202846
 - Victim order: 1781.3158213969338 SOL filled

* Move market down (dump purchased SOLs)
 - Final mark price: 97.2345550062
 - Victim final collateral:  999822.114058
 - Hacker final collateral: 1008193.880764

* Withdraw hacker funds
- Hacker profit: $8193
```

Example log output from our Proof of Concept.

We could not test the PoC against a forked mainnet deployment of Drift, as currently, stop and limit orders remain a devnet-only beta feature.

Because Solana transactions may include multiple instructions, the sandwich attack can be performed atomically, eliminating the window for arbitrageurs to arb the market back down. Similarly, it also avoids issues with funding payments while performing the sandwich.

During the development of the PoC, we calculated the optimal manipulation ratios for the attacker given specific market conditions. Generally, the attacker's profit depends on the total volume of force-fillable orders and their limit prices. We found that there is a “sweet spot”, as too much manipulation increases costs in fees, whereas too little manipulation reduces profits as the curve’s slope is less steep closer to the center. However, if the victim order’s trigger price lies above the optimal price, this may impact attacker profits.



Example of an attacker’s profit curve on Devnet deployment. In this specific instance, the optimal ratio was 4.72% above current market price.

Recommendations

It is important for Drift to be sensitive to market volatility. However, using the raw mark price without additional validations can be problematic. We recommend validating the mark price against the mark TWAP and the oracle price. If a trigger order fill would fail under the mark TWAP and the oracle price, but succeed under the raw mark price, we suggest blocking the order fill as the current market conditions may be unhealthy.

Remediation

This class of attack has been mitigated by checking that, if a valid oracle exists, that the oracle agrees with the trigger price within 1%. This check was applied in commit 7f5df1c5.

3.2. An attacker may liquidate users unfairly with AMM manipulation

- **Target:** ClearingHouse
- **Severity:** High
- **Impact:** Medium
- **Category:** Business Logic
- **Likelihood:** Medium

Description

The AMM calculates a user's margin ratio based on their position sizes and the current AMM mark price. The AMM mark price may be manipulated by an attacker as this price is used without TWAP or oracle validation. This would allow an attacker to manipulate the AMM (up to 10% away from the oracle) and liquidate users unfairly, possibly profiting in the process.

This is similar to the previous finding.

Attack Scenario

According to our calculations, users who exceed these margin ratios can be partially or fully liquidated with a ±10% mark price manipulation:

	Long	**Short**
Partial Liquidation	≥ 6.4x	≤ -5.9x
Full Liquidation	≥ 6.9x	≤ -6.5x

These margin ratios exceed the initial margin limit of 5x for opening a position, but would not be unusual in a volatile market. Since the attacker earns a rebate when liquidating users, this attack can be profitable if the attacker liquidates a large total position after manipulating the market. Furthermore, the taker fee of 0.1% incurred by the attacker to perform the manipulation is relatively low compared to the liquidation rebates.

Impact

An attacker may be able to liquidate users at an unfair price that is not representative of true market conditions.

Recommendations

It is important for Drift to be sensitive to market volatility. However, using the raw mark price without additional validations can be problematic. We recommend validating the mark price against the mark TWAP and the oracle price. If both the mark TWAP and oracle price would not lead to a liquidation, but the raw mark price would, we recommend blocking the liquidation as the current market conditions may be unhealthy.

Remediation

This class of attack has been mitigated by checking that, if a valid oracle exists, that the oracle agrees with the trigger price within 1%. This check was applied in commit 7f5df1c5.

3.3. Potential integer overflow in liquidate()

- **Target:** ClearingHouse
- **Severity:** Low
- **Impact: Low**
- **Category:** Coding Mistakes
- **Likelihood:** Low

Description

In liquidate, `base_asset_value` and `base_asset_value_to_close` are directly added to `base_asset_value_closed` without checking for integer overflow.

```
pub fn liquidate(ctx: Context<Liquidate>) -> ProgramResult {
...
            let base_asset_amount = base_asset_amount.unsigned_abs();
            // Unchecked arithmetic operation
            base_asset_value_closed += base_asset_value;
...
            .ok_or_else(math_error!())?;
            // Unchecked arithmetic operation
            base_asset_value_closed += base_asset_value_to_close;
...
```

Unchecked addition arithmetic

Impact

The variable `base_asset_value_closed` is not directly used for any calculations, but simply as a record-keeping mechanism in LiquidationHistory. While this issue poses no security risk to Drift at present, an overflow could potentially break an off-chain entity using LiquidationHistory as an information source.

Recommendations

Use `checked_add` for the addition operation instead.

Remediation

The issue has been fixed in commit 4acb72c8

3.4. Anchor IDL instructions are enabled

- **Target:** ClearingHouse
- **Severity:** Low
- **Impact:** Informational
- **Category:** Code Maturity
- **Likelihood:** Low

Description

Anchor, by default, transparently injects on-chain IDL instructions to every contract. Their purpose is to store the IDL at a deterministic address on-chain for easier retrieval by clients. The IDL account is a PDA and its management is handled through these instructions. If uninitialized, it would allow any entity to create and become the authority of the IDL, controlling most of the content as it allows arbitrary buffers to be stored.

Impact

PDAs and other program-owned accounts are often considered trusted and verified, as usually only the program can write to them. Thus, many contracts use ownership as a trust boundary. Problems can arise when these assumptions do not hold, as an attacker could store arbitrary data in an account owned by a trusted program.

Although every structure in Anchor is stored with an 8 byte discriminator that prevents deserialization of malicious data and type confusion, this may pose a composability risk for downstream consumers of the Drift API if they assume account ownership represents a security boundary.

Lastly, unnecessary on-chain features always pose a risk, as they increase the attack surface of the program.

Recommendations

Disable IDL Instructions by using the `no-idl` compiler flag.

Remediation

Since the Anchor IDL account is initialized and Drift plans to use it to improve integration with Solana blockchain explorers, this issue requires no remediation.

3.5. Rounding of signed division can be potentially unfavorable

- **Target:** ClearingHouse
- **Severity:** Low
- **Impact: Low**
- **Category:** Business Logic
- **Likelihood:** Low

Description

The functions `settle_funding_payment` and `calculate_funding_payment_in_quote_ precision` perform a precision-reducing division of a signed value. This rounds the quotient towards zero: positive numbers are rounded down (floored), while negative numbers are rounded up (ceiled).

```
pub fn settle_funding_payment(...) -> ClearingHouseResult
    // ...
    let funding_payment_collateral = funding_payment
        .checked_div(AMM_TO_QUOTE_PRECISION_RATIO_I128)
        // ^ Loss of precision here
        .ok_or_else(math_error!())?;
    // Adds funding_payment_collateral to user.collateral
    user.collateral = calculate_updated_collateral(user.collateral,
funding_payment_collateral)?;

   Ok(())
}
```

More concretely, negative funding payments (i.e., the user *owes* money) would be rounded towards zero, reducing the amount owed.

Impact

A user may be able to shave off miniscule amounts from negative funding balances.

Recommendations

Add a checked_sub(1) if the funding payment is negative so that users who owe funding do not get an extra +1 from integer division. This pattern is used elsewhere in the code base already, such as in the function `calculate_pnl`.

Remediation

The issue has been acknowledged by Drift, and a fix is pending.

3.6. Multiple internal inconsistencies in Anchor macros

- **Target:** ClearingHouse
- **Severity:** Low
- **Impact:** Informational
- **Category:** Code Maturity
- **Likelihood:** N/A

Description

The usage of anchor macros for access control is inconsistent across ClearingHouse. For example:

```
pub struct ClosePosition<'info> {
    #[account(mut)]
    pub state: Box<Account<'info, State>>,
    #[account(
        mut,
        has_one = authority,
        // user_positions is checked here
        constraint = &user.positions.eq(&user_positions.key())
    )]
    pub user: Box<Account<'info, User>>,
    pub authority: Signer<'info>,
    #[account(
        mut,
        constraint = &state.markets.eq(&markets.key())
    )]
    pub markets: AccountLoader<'info, Markets>,
    #[account(
        mut,
        // ... instead of being checked here [1]
        has_one = user
    )]
    pub user_positions: AccountLoader<'info, UserPositions>,
    #[account(
        mut,
        constraint = &state.trade_history.eq(&trade_history.key())
    )]
    pub trade_history: AccountLoader<'info, TradeHistory>,
```

Inconsistent authorization checks

We believe that each account in the struct should be verified in its own `#[account]` macro. For example, `user_positions` should be checked against the `user` account at `[1]`. This is present in several other structures, such as `CancelOrder`, `FillOrder`, etc.

Impact

A manual review found no security issues with the current implementation. However, while there is no immediate impact, inconsistencies like these make the code difficult to read and reason about, which could lead to future bugs.

Recommendations

Refactor the `#[account]` macro checks to be more consistent.

Remediation

The issue has been acknowledged by Drift. It is believed that no changes are necessary at this time.

3.7. Use of unchecked division operations

- **Target:** ClearingHouse
- **Severity:** Low
- **Impact:** Informational
- **Category:** Coding Mistakes
- **Likelihood:** Low

Description

The functions `limit_price_satisfied` and `asset_to_reserve_amount` use unchec- ked division operations.

```
pub fn limit_price_satisfied(
   limit_price: u128,
   quote_asset_amount: u128,
   base_asset_amount: u128,
   direction: PositionDirection,
) -> ClearingHouseResult<bool> {
   let price = quote_asset_amount
       .checked_mul(MARK_PRICE_PRECISION *
AMM_TO_QUOTE_PRECISION_RATIO)
       .ok_or_else(math_error!())?
       .div(base_asset_amount);
...

pub fn asset_to_reserve_amount(
   quote_asset_amount: u128,
   peg_multiplier: u128,
) -> ClearingHouseResult<u128> {
   Ok(quote_asset_amount
       .checked_mul(AMM_TIMES_PEG_TO_QUOTE_PRECISION_RATIO)
       .ok_or_else(math_error!())?
       .div(peg_multiplier))
}
```

Use of unchecked division operations

Impact

Although this issue does not pose a security risk at present, we believe that it is important to follow best practices and use checked arithmetic operations whenever possible.

Recommendations

Use `checked_div` instead of `div`.

Remediation

The issue has been acknowledged by Drift and a fix is pending.

4. Discussion

In this section, we discuss miscellaneous interesting observations during the audit that are noteworthy and merit some consideration.

1. We carefully reviewed all Anchor `Accounts` structs in context.rs, and did not find any access control issues. Additionally, we checked all program instructions for code coverage, and found that there were tests for all instructions, except one (`initialize_user_with_explicit_payer`). We applaud Drift for their attention to detail and safe coding practices.

2. We carefully reviewed all instances of integer division in the program, and did not find any issues with conditioning and numerical stability. For instance, we checked that any operations with rational operands multiplied the numerator before dividing the denominator. However, conducting this review was tricky and required above-average attention to detail.

   We believe that this class of error could be a potential source of future bugs, and recommend using a unified Rational class rather than separate numerator and denominator values. The Rational type could also be parameterized by a precision value. This would further obviate the possibility of numerical bugs by preventing one from accidentally mixing values with different decimals. Such a design would also greatly simplify conversions between different precisions.

3. We reviewed instances of integer division for instances of rounding that are potentially unfavorable to Drift. Aside from one minor finding (detailed elsewhere in this document), we found no obvious instances of this class of error. However, we recommend remaining vigilant for any future instances of this bug class, as such errors can be potentially catastrophic.

5. Appendix A: ClearingHouse Fuzzer

We developed a fuzzer leveraging the poc-framework. It forks Drift's mainnet state and sends random instructions to the cloned program while checking certain business logic invariants. The fuzzer checked for the following:

1. Sequence of random `deposit_collateral` and `withdraw_collateral`:
   - An attacker should never end up with more USDC than they started with.
2. Sequence of random `open_position` and `close_position`:
   - An attacker should never end up with more collateral than they started with.
   - An attacker should never end up with a position with >5x leverage.

As part of this engagement, we provided the fuzzing framework to Drift, and we recommend integrating fuzz tests with the existing test suite. Fuzzers expand code coverage and decrease the likelihood of bugs. This is because fuzzers regularly catch corner cases that human programmers fail to consider when writing unit tests. Another benefit of fuzz testing is that developers can stress test business-critical invariants that are specific to the application. This benefits developer confidence, making the development lifecycle both faster and more secure.