Juiced

Audit

Presented by:

OtterSec contact@osec.io

Harrison Green hgarrereyn@osec.io

William Wang defund@osec.io

Contents

01 | Executive Summary

Overview

Juiced engaged OtterSec to perform an assessment of the `juiced` program.

This assessment was conducted between July 15th and July 29th, 2022.

Critical vulnerabilities were communicated to the team prior to the delivery of the report to speed up remediation. After delivering our audit report, we worked closely with the team over to streamline patches and confirm remediation.

We delivered final confirmation of the patches July 29th, 2022.

Key Findings

The following is a summary of the major findings in this audit.

- 7 findings total
- 3 vulnerabilities which could lead to loss of funds
  - OS-JUI-ADV-00: Resolved
  - OS-JUI-ADV-01: Resolved
  - OS-JUI-ADV-02: Resolved

As part of this audit, we also provided proofs of concept for each vulnerability to prove exploitability and enable simple regression testing. These scripts can be found at osec.io/pocs/juiced. For a full list, see Appendix B.

02 | Scope

The source code was delivered to us in a git repository at github.com/juiced-fi/juiced-protocol. This audit was performed against commit `f7c035f`.

There was 1 program included in this audit. A brief description for each program is given below. A full list of program files and hashes can be found in Appendix A.

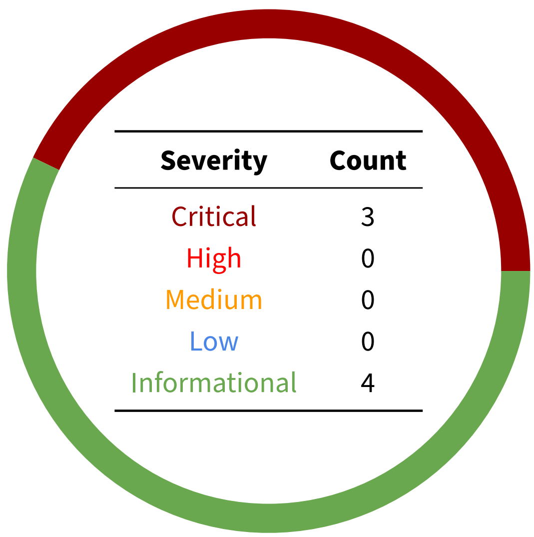
Name	Description
juiced	Staking pool with a variety of strategies on Mango and Mercurial.

03 | Findings

Overall, we report 7 findings.

We split the findings into **vulnerabilities** and **general findings**. Vulnerabilities have an immediate impact and should be remediated as soon as possible. General findings don't have an immediate impact but will help mitigate future vulnerabilities.

The below chart displays the findings by severity.



Severity	Count
Critical	3
High	0
Medium	0
Low	0
Informational	4

Proofs of Concept

For each vulnerability we created a proof of concept to enable easy regression testing. We recommend integrating these as part of a comprehensive test suite. The proof of concept directory structure can be found in Appendix B.

A GitHub repository containing these proof of concepts can be found at osec.io/pocs/juiced.

To run a POC:

```sh
./run.sh <directory name>
```

For example,

```sh
./run.sh os-jui-adv-00
```

Each proof of concept comes with its own patch file which modifies the existing test framework to demonstrate the relevant vulnerability. We also recommend integrating these patches into the test suite to prevent regressions.

04 | Vulnerabilities

Here we present a technical analysis of the vulnerabilities we identified during our audit. These vulnerabilities have **immediate** security implications, and we recommend remediation as soon as possible.

Rating criteria can be found in Appendix E.

ID	Severity	Status	Description
OS-JUI-ADV-00	Critical	Resolved	The USDC sweeper vault is not necessarily used during deposit and withdrawal.
OS-JUI-ADV-01	Critical	Resolved	The USDC and BTC/SOL RootBank accounts are not necessarily used during deposit and withdrawal.
OS-JUI-ADV-02	Critical	Resolved	Mango-exclusive instructions incorrectly calculate notional value when invoked with Mercurial strategies.

OS-JUI-ADV-00 [crit] [Resolved] | Insufficiently constrained token vaults

In addition to Mango/Mercury accounts, every Juiced carton stores unused USDC in a sweeper vault. This is also where users deposit and withdraw funds in exchange for pool tokens, which represent a fractional share of the carton's notional value.

On deposit, the protocol mints pool tokens to the depositor such that the ratio between pool tokens and notional value remains constant. Naturally, withdrawal requires burning pool tokens; the protocol then transfers USDC in order to maintain the ratio.

The issue is that the `deposit` and `withdraw` instructions do not properly validate the `vault` account that is passed into their contexts. Any USDC token account owned by the program's `authority` PDA will be accepted, even though `vault` should always be the carton's sweeper account.

instructions/mango/deposit.rs RUST

```rust
#[account(
    mut,
    constraint = (vault.mint == owner_usdc_account.mint && vault.owner ==
     ↪  authority.key())
)]
pub vault: Box<Account<'info, TokenAccount>>,
```

By providing a different account, an attacker can misrepresent the notional value of the sweeper vault — and hence the overall carton — during deposit and withdrawal. This can easily be leveraged into loss of funds.

utils.rs RUST

```rust
fn calculate_sweeper_notional(
    mango_group: &MangoGroup,
    sweeper_account: &TokenAccount,
) -> Option<I80F48> {
    let sweeper_holdings = sweeper_account.amount;
    let decimals = mango_group.tokens[QUOTE_INDEX].decimals;
    bits_to_token(sweeper_holdings, decimals)
}
```

An identical bug is present in the `deposit_mercurial` and `withdraw_mercurial` instructions, where the sweeper vault is named `juiced_usdc_vault`.

Proof of Concept

Suppose a Juiced carton has 10,000 USDC in the sweeper vault and 10,000 minted pool tokens. Consider the following attack:

1. The attacker creates a token account holding 0.01 USDC. This will function as the fake sweeper vault.
2. The attacker invokes `deposit` with the fake vault, and transfers 100 USDC. The protocol mistakenly calculates the notional value to be 0.01 USDC. This corresponds with 10,000 pool tokens, so it mints 100,000,000 pool tokens to the attacker.
3. The attacker invokes `withdraw` with the real vault, and burns 100,000,000 pool tokens. The protocol calculates the notional value to be 10,000 USDC. This corresponds with 100,010,000 pool tokens, so it transfers $\approx$ 9,999 USDC to the attacker.

The attacker's net profit is $\approx$ 9,899 USDC.

Remediation

In the `DepositWithdraw` and `DepositWithdrawMercurial` contexts, verify that the provided sweeper vault address matches `juiced.usdc_vault_key`.

instructions/mango/deposit.rs DIFF

```diff
@@ -45,7 +45,7 @@ pub struct DepositWithdraw<'info> {

     #[account(
         mut,
-        constraint = (vault.mint == owner_usdc_account.mint && vault.owner ==
    ↪   authority.key())
+        address = juiced.usdc_vault_key,
     )]
     pub vault: Box<Account<'info, TokenAccount>>,
```

Patch

Fixed in #572.

OS-JUI-ADV-01 [crit] [Resolved] | Insufficiently constrained RootBank accounts

In Mango, each asset has an associated RootBank which holds the interest rate parameters for depositors and borrowers. The compounded interest rates are stored in the `deposit_index` and `borrow_index` fields, which Juiced uses to calculate the notional value of a Mango account.

utils.rs RUST

```rust
let usdc_deposit_index = usdc_root_bank.deposit_index.checked_div(bits_mult)?;
let usdc_borrow_index = usdc_root_bank.borrow_index.checked_div(bits_mult)?;
let token_deposit_index = token_root_bank.deposit_index.checked_div(bits_mult)?;
let token_borrow_index = token_root_bank.borrow_index.checked_div(bits_mult)?;

let token_deposits =
    ↪ mango_account.deposits[symbol_index].checked_mul(token_deposit_index)?;
let token_borrows =
    ↪ mango_account.borrows[symbol_index].checked_mul(token_borrow_index)?;
let total_token = token_deposits.checked_sub(token_borrows)?;
msg!("mango token total: {:?}", total_token.to_string());

let usdc_deposits =
    ↪ mango_account.deposits[QUOTE_INDEX].checked_mul(usdc_deposit_index)?;
let usdc_borrows =
    ↪ mango_account.borrows[QUOTE_INDEX].checked_mul(usdc_borrow_index)?;
let total_usdc = usdc_deposits.checked_sub(usdc_borrows)?;
msg!("mango total usdc: {:?}", total_usdc.to_string());
```

The issue is that the `deposit` and `withdraw` instructions do not properly validate the `usdc_root_bank` and `token_root_bank` accounts that are passed into their contexts. Although the `load_mango_data` function requires them to be proper RootBank accounts owned by the Mango program, their corresponding assets are not verified to be USDC and BTC/SOL.

instructions/mango/deposit.rs RUST

```rust
/// CHECK: mango usdc root bank
pub usdc_root_bank: UncheckedAccount<'info>,

/// CHECK: mango token root bank
pub token_root_bank: UncheckedAccount<'info>,
```

By providing an unexpected RootBank, an attacker can misrepresent the notional value of the Mango account — and hence the overall carton — during deposit and withdrawal. This can easily be leveraged into loss of funds, especially if one initializes a RootBank with extreme interest rates. Note that this bug also impacts the `deposit_mercurial` and `withdraw_mercurial` instructions.

Proof of Concept

Suppose a Juiced carton has 5,000 USDC in the sweeper vault, 5,000 USDC in the Mango account, and 10,000 minted pool tokens. Consider the following attack:

1. The attacker invokes `deposit` with the wBTC RootBank instead of the USDC RootBank, and transfers 10,000 USDC. The protocol mistakenly calculates the notional value to be $\approx$ 9,841 USDC. This corresponds with 10,000 pool tokens, so it mints $\approx$ 10,161 pool tokens to the attacker.
2. The attacker invokes `withdraw` with the correct USDC RootBank, and burns $\approx$ 10,161 pool tokens. The protocol calculates the notional value to be $\approx$ 20,000 USDC. This corresponds with 20,161 pool tokens, so it transfers $\approx$ 10,080 USDC to the attacker.

The attacker's net profit is $\approx$ 80 USDC.

Remediation

Verify that `usdc_root_bank` is the USDC root bank and `token_root_bank` is the BTC or SOL root bank (depending on the Juiced strategy). One option is to store the expected addresses in the Juiced account data. Another option is to retrieve them from the `mango_group` account, which is already verified properly.

Patch

Fixed in #576.

OS-JUI-ADV-02 [crit] [Resolved] | Mango-exclusive instructions accept Mercurial strategies

There are two categories of Juiced strategies: Mango-exclusive (stores assets in sweeper vault and Mango account) and Mercurial (additionally stores assets in Mercurial account). When staking funds in Mercurial strategies, users are supposed to invoke the `deposit_mercurial` and `withdraw_mercurial` instructions. This is because they must include Mercurial assets when calculating the Juiced carton's notional value.

utils.rs RUST

```rust
let sweeper_and_mango = calculate_juiced_notional(
    juiced,
    mango_account,
    mango_group,
    mango_cache,
    usdc_root_bank,
    token_root_bank,
    sweeper_account,
    is_withdrawal,
)?;
msg!("sweeper_and_mango: {}", sweeper_and_mango.to_string());
let mercurial_amount = calculate_mercurial_notional(
    lp_token_account,
    mercurial_lp_mint,
    mercurial_vault,
    is_withdrawal,
)?;
msg!("mercurial_amount: {}", mercurial_amount.to_string());
mercurial_amount.checked_add(sweeper_and_mango)
```

The issue is that the `deposit` and `withdraw` instructions, which expect Mango-exclusive strategies, may be invoked with Mercurial strategies. The program assumes the Mercurial account does not exist, and hence underestimates the carton's notional value. Similar to OS-JUI-ADV-00 and OS-JUI-ADV-01, this introduces an arbitrage opportunity which can be leveraged into loss of funds.

Proof of Concept

Suppose a Juiced carton has 5,000 USDC in the sweeper vault, $\approx$ 4,995 USDC in the Mercurial account, and 10,000 minted pool tokens. Consider the following attack:

1. The attacker invokes `deposit` and transfers 10,000 USDC. The protocol mistakenly calculates the notional value to be 5,000 USDC. This corresponds with 10,000 pool tokens, so it mints 20,000 pool tokens to the attacker.

2. The attacker invokes `withdraw_mercurial` and burns 20,000 pool tokens. The protocol calculates the notional value to be 19,995 USDC. This corresponds with 30,000 pool tokens, so it transfers $\approx$ 13,330 USDC to the attacker.

The attacker's net profit is $\approx$ 3,330 USDC.

Remediation

In the `deposit` and `withdraw` instructions, explicitly require the strategy to be `BtcMangoFunding` or `SolMangoFunding`. In the `deposit_mercurial` and `withdraw_mercurial` instructions, explicitly require the strategy to be `BtcMangoMercurial` or `SolMangoMercurial`.

Patch

Fixed in #579.

05 | General Findings

Here we present a discussion of general findings during our audit. While these findings do not present an immediate security impact, they do represent antipatterns and could introduce a vulnerability in the future.

ID	Description
OS-JUI-SUG-00	Enforce stricter verification on admin-gated functionality.
OS-JUI-SUG-01	Maintain liquidity in the sweeper vault.
OS-JUI-SUG-02	Use more specialized Anchor constraints whenever possible.
OS-JUI-SUG-03	Use consistent naming across accounts and contexts.

OS-JUI-SUG-00 | Admin-gated functionality

Description

Although the `initialize` instruction is intended for the Juiced backend, it does not compare the provided `backend` signer against a fixed public key. If the backend does not initialize all strategies immediately after deployment, an attacker can take control of one.

Many admin-gated instructions also fail to perform sufficient account checks. If the backend makes a mistake, this could lead to invalid state.

Remediation

In the `initialize` instruction, verify that the `backend` account matches a fixed public key. To simplify testing, this constraint should only apply on mainnet.

Consider incorporating the following (non-comprehensive) list of account checks:

- In the `Initialize` context, `usdc_token` and `mango_program` should be fixed constants on mainnet.
- In the `InitializeMercurial` context, `usdc_token` and `mercurial_program` should be fixed constants on mainnet. Also, `mercurial_lp_token` should be equal to `mercurial_vault.lp_mint`.
- In the `MoveFromMango` and `MoveToMango` contexts, `vault` should be equal to `juiced.usdc_vault_key`.
- In the `SettleSpotFunds` context, `juiced_usdc_vault` should be equal to `juiced.usdc_vault_key`.
- In the `MoveToFromMercurial` context, `juiced_usdc_vault` should be equal to `juiced.usdc_vault_key`.

OS-JUI-SUG-01 | Limited withdraw capabilities

Description

In order to generate yield, the Juiced protocol deposits funds into Mango and Mercurial. However, this means that users can only withdraw up to the amount currently held in the sweeper vault. During periods of large withdrawal, the carton may not be able to service every request.

Remediation

If the sweeper vault's balance dips below a certain threshold, the Juiced backend should move back assets from Mango and Mercurial. Alternatively, one could modify the on-chain program to automatically CPI into these programs, if necessary during withdrawal.

OS-JUI-SUG-02 | Anchor constraints

Description

The Juiced program often uses `constraint`, which is generic, when a specialized Anchor constraint is available. For example, the `has_one` and `address` constraints are more suitable for validating account addresses.

instructions/mango/deposit.rs DIFF

```diff
@@ -64,7 +64,7 @@ pub struct DepositWithdraw<'info> {

     #[account(
         mut,
-        constraint = (juiced.pool_token == pool_mint.key())
+        address = juiced.pool_token,
     )]
     pub pool_mint: Box<Account<'info, Mint>>,
 }
```

Similarly, SPL Token constraints are more suitable for validating SPL Token accounts.

instructions/mango/deposit.rs DIFF

```diff
@@ -51,7 +51,8 @@ pub struct DepositWithdraw<'info> {

     #[account(
         mut,
-        constraint = (owner_usdc_account.mint == vault.mint &&
     ↪  owner_usdc_account.owner == owner.key())
+        token::mint = vault.mint,
+        token::authority = owner,
     )]
     pub owner_usdc_account: Box<Account<'info, TokenAccount>>,
     pub token_program: Program<'info, Token>,
```

Remediation

- Use the `has_one` or `address` constraints to match public keys.
- Use the `token::mint` and `token::authority` constraints to validate a token account.

OS-JUI-SUG-03 | Use consistent naming

Description

Accounts are often inconsistently named across context and struct definitions, which produces less readable code. For example, the `JuicedState` struct has a field named `usdc_vault_key`, which holds the sweeper vault's public key. In the `DepositWithdraw` context, it is named `vault`. In the `DepositWithdrawMercurial` context, it is named `juiced_usdc_vault`.

Remediation

Whenever possible, refer to an account by the same name across contexts and structs.

A | Program Files

Below are the files in scope for this audit and their corresponding SHA256 hashes.

```
Cargo.toml                          d81fd0f1c7de601741c5d662719f1df6d370492651fed2cfcdf66b28e6fa045c
Xargo.toml                          815f2dfb6197712a703a8e1f75b03c6991721e9eb7c40dfaec8b0b49da4aa629
build-me.txt                        9fa8637ca8ab48d2f6a3f3d43bbd6941b6a02786b50e6e87ac267039821f7b31
src
  errors.rs                         8bf60aa33b6176b33bcb0295fcffe7e767326d35ca43439ca9a10665cc4c12ad
  lib.rs                            ec0ff64c0921cd0eabfd69ac47c1100e689d18754fc893db7ce33a935b4f820b
  utils.rs                          98e311f020e3e7ab98bb6bee7cb5b06772c927a5cf627832b1d7d691b86bf337
  instructions
    mod.rs                          5cd3ce2cd1767e2bc224fcffde5ac0e3615bc83540a842657eb9edd577c51c5c
    mango
      cancel_all_perp_orders.rs     f1b85d8430051608b9d82b66789db644c3e20547fea8443ce9927627dfa37aa0
      cancel_all_spot_orders.rs     310ceaf67dbb6b66a08d650b15885f274ac4b53006b3ae725782d2a897038737
      cancel_perp_order.rs          2f9f38e1873c88ecce24eb5a736528432431e720835d34d6690ef7b84027398a
      cancel_spot_order.rs          dfe4e801e7f1bf8b4cbbf63f975976079cf112d193c1259980d72003baefadbb
      create_spot_open_orders.rs    fae892a19543ddaab767bc0158dd00f7435605e18eec4fe96f180b3763016e04
      deposit.rs                    6189e8ec4f5d7c64724706e1c894541e11599a1172ea463807294163cb6c6b77
      init.rs                       02e5b77d0d158db9c7252c277a2888b690a27725a218b1208934ae5b665b04ae
      mod.rs                        b575319a2cec245d612214d83dde90da1cd735e199795767da918621c313186f
      move_from_mango.rs            730f5e86c8f39b762282b4fc32c2c5fcf32e2a544413091ecec15985a648a455
      move_to_mango.rs              b96bb315165a71a038e6992441dc2e5f35c9c2717ed89beb7ff54b04dfc108d2
      settle_spot_funds.rs          2049d69aa2a8d4f41a2dd11bccec9ec4a447074dca5b1f19616d014f497679cd
      trade_perp.rs                 3d025a66e2a11e9aeb41f0b7a0596e1e14c44ee31ae8cf3218670dee8782078a
      trade_spot.rs                 b538864167e6ccbbac7eed8d42e2914339f879ec78e7734b47c9814b75644717
      withdraw.rs                   7f25d9464f63df2a7415715b33a11fc9ece864bca61823e6a9c4ae1bc81a83d8
    mercurial
      deposit.rs                    b4430d71396afff418aff2bd12d39bb0ffa38b37c3c9b83aaf621929d85e6fc1
      init_mercurial.rs             c821e9b66cd73d9c8fe63d77142d789db627c809c36b2a148905948fec9da0de
      mod.rs                        4ee31e32343bc32cbb2f6b050703df6954871afc1b30a3a4490988b6bd653769
      move_from_mercurial.rs        3b96ac53da035d7d3f23e7f9a4bcbb7008ed984c72f647f9e2abffc51db3bd93
      move_to_mercurial.rs          49dd5cd3277daac8dd612da94bf8f2a77908b6fd388cfa1f49ac895a30cc12e3
      withdraw.rs                   1e3e301aa5911f8ed99c2bb251326062bb89e2227d760ee27aa89783b7d6885b
  structs
    juiced.rs                       2d914f4d711c9a2e24e47e2b491b85575da48373379d8e5a7266f37b639ddd85
    mango.rs                        7ea7bd42adc98c60bcf1b2603b7d39ffc6cc457685933fc4c9df37a6b9a8e1d9
    mercurial.rs                    342e403dcf9b09406b9fb724ca2153bfc964274cc84f4fbcad46b02a7b05fb39
    mod.rs                          da0b53fb6ad156ee43f79a9a83e35bf16ed717141c11b7f13be0d46515b0640e
```

B | Proofs of Concept

Below are the provided proof of concept files and their corresponding SHA256 hashes.

```
Dockerfile                   c58e7b1a89addf088e70cf4b1abb7ffcc708600ce08f859e7069db03b2efdc98
README.md                    de95a89ba81cb1397699415362b701cf52768f37df53f752be427abc5886c4cd
run.sh                       e1bb4cc229f6f40481997c25cc38bf23486615b612479303036ef9dbf634dee5
pocs
  Cargo.toml                 5948dca867fc75d34fdcf7286e1dce02eb4ad30f7c19a7f0ae7419d0a1de7c4a
  mango_cache.dat            db9865b4d7f77712b8844d438cba2365e34010644f8bcaeb2d1d5fa4b76d2e57
  mango_group.dat            f1c67c1560739d02592e532481266465cf9ceca11978945bc56205f8d2585c52
  mango_node_usdc.dat        a836037874fc6d6add01d144a20b34517b3149df24a54a88de4a956f58321d5d
  mango_root_usdc.dat        9b215324b36eb4a9e3f545cc8a8dd61208f840de94a5d32e4fe5e0dadae7e443
  mango_root_wbtc.dat        76d3e629170d140208c7ce9caebf33a82e1950715eb6873c697f887b16bd8356
  mango_vault_usdc.dat       ae06d3b5ef3d779eb96512ca6d37fd5469cca2cc37ce09a62d4b1b182d9a2ac7
  mercurial_lp_mint.dat      57666d5acd36acb24364a9f801adc2b81032ca06c1adf4358d34e7235dc5c307
  mercurial_usdc_vault.dat   c2d9cdcdac0171c78a6e3ae59c8dd890970e3ca74f3604a0c8917a34004c78fd
  mercurial_vault.dat        ab1ff3a2c45da53123cc929b1d3deb93af465b20c7cbc13f36cfa3d336801f1a
  usdc_mint.dat              5ac48b8a37ddb9a5b8cf1ab6876755475266358879094caa6ec0efabba63a438
  prog
    juiced_mainnet.so        b70d2db9cea946c51e144c3828ab0a0999aa56a3ba4c207703bca25d20c4116d
    mango_mainnet.so         dbb567daab96e04961bbbe372098f95a72199346260596d74b79c6b687029d59
    mercurial_mainnet.so     378903d85e4d62a4fc0faeb654ac9646c280d8c63a95be5d0b3affa2599d226b
  src
    adv_00.rs                e82ee7b5058cc9f4062622c1a7696ef8c3dd78e44f78c60efe5baeed63489423
    adv_01.rs                f5c0fec52435febf71ba50b1fa81039527ade699628dd31c1ee26992eaf57df8
    adv_02.rs                49cdae2cbcdcb57b228934334cc96ebd469fd553d342fc1288ef563920a6eb11
    lib.rs                   37f9e46c18405f3016b28fb6d12a1fee831c294075a166f738f77c6ba075def9
```


C | Procedure

As part of our standard auditing procedure, we split our analysis into two main sections: design and implementation.

When auditing the design of a program, we aim to ensure that the overall economic architecture is sound in the context of an onchain program. In other words, there is no way to steal tokens or deny service, ignoring any Solana specific quirks such as account ownership issues. An example of a design vulnerability would be an onchain oracle which could be manipulated by flash loans or large deposits.

On the other hand, auditing the implementation of the program requires a deep understanding of Solana's execution model. Some common implementation vulnerabilities include account ownership issues, arithmetic overflows, and rounding bugs. For a non-exhaustive list of security issues we check for, see Appendix D.

Implementation vulnerabilities tend to be more "checklist" style. In contrast, design vulnerabilities require a strong understanding of the underlying system and the various interactions: both with the user and cross-program.

As we approach any new target, we strive to get a comprehensive understanding of the program first. In our audits, we always approach any target in a team of two. This allows us to share thoughts and collaborate, picking up on details that the other missed.

While sometimes the line between design and implementation can be blurry, we hope this gives some insight into our auditing procedure and thought process.

D | Implementation Security Checklist

Unsafe arithmetic

Integer underflows or overflows	Unconstrained input sizes could lead to integer over or underflows, causing potentially unexpected behavior. Ensure that for unchecked arithmetic, all integers are properly bounded.
Rounding	Rounding should always be done against the user to avoid potentially exploitable off-by-one vulnerabilities.
Conversions	Rust as conversions can cause truncation if the source value does not fit into the destination type. While this is not undefined behavior, such truncation could still lead to unexpected behavior by the program.

Account security

Account Ownership	Account ownership should be properly checked to avoid type confusion attacks. For Anchor, the safety of unchecked accounts should be clearly justified and immediately obvious.
Accounts	For non-Anchor programs, the type of the account should be explicitly validated to avoid type confusion attacks.
Signer Checks	Privileged operations should ensure that the operation is signed by the correct accounts.
PDA Seeds	PDA seeds are uniquely chosen to differentiate between different object classes, avoiding collision.

Input validation

Timestamps	Timestamp inputs should be properly validated against the current clock time. Timestamps which are meant to be in the future should be explicitly validated so.
Numbers	Sane limits should be put on numerical input data to mitigate the risk of unexpected over and underflows. Input data should be constrained to the smallest size type possible, and upcasted for unchecked arithmetic.
Strings	Strings should have sane size restrictions to prevent denial of service conditions
Internal State	If there is internal state, ensure that there is explicit validation on the input account's state before engaging in any state transitions. For example, only open accounts should be eligible for closing.

Miscellaneous

Libraries	Out of date libraries should not include any publicly disclosed vulnerabilities
Clippy	cargo clippy is an effective linter to detect potential anti-patterns.

E | Vulnerability Rating Scale

We rated our findings according to the following scale. Vulnerabilities have immediate security implications. Informational findings can be found in the General Findings section.

Critical	Vulnerabilities which immediately lead to loss of user funds with minimal preconditions Examples: • Misconfigured authority/token account validation • Rounding errors on token transfers
High	Vulnerabilities which could lead to loss of user funds but are potentially difficult to exploit. Examples: • Loss of funds requiring specific victim interactions • Exploitation involving high capital requirement with respect to payout
Medium	Vulnerabilities which could lead to denial of service scenarios or degraded usability. Examples: • Malicious input cause computation limit exhaustion • Forced exceptions preventing normal use
Low	Low probability vulnerabilities which could still be exploitable but require extenuating circumstances or undue risk. Examples: • Oracle manipulation with large capital requirements and multiple transactions
Informational	Best practices to mitigate future security risks. These are classified as general findings. Examples: • Explicit assertion of critical internal invariants • Improved input validation • Uncaught Rust errors (vector out of bounds indexing)