# DeFindex Vault: Fee Prediction via Simulation

## Main Objective

**Calculate how many fees will be locked in a future deposit or withdrawal.**

### Why is this important?

The vault's `total_managed_funds` function returns **all funds including locked fees**. To understand the real available funds and predict fee impact, you need to know how many fees would be locked if someone interacted with the contract right now.

### The Technique

We use a **simulation technique** that:

1. Simulates calling `report()` to update strategy performance
2. Simulates calling `lock_fees()` immediately after
3. Returns the **exact fee amounts** that would be locked at this moment
4. Does NOT actually execute these operations on-chain (read-only simulation)

This gives you a **snapshot of pending fees** without modifying the contract state.

## What This Demo Does

1. **Dynamically retrieve the vault manager** using `get_manager()`
2. **Simulate atomic execution** of `report()` + `lock_fees()` to calculate pending fees
3. **Return fee amounts** that will be locked on the next user interaction

## Why Use the Router?

**Fees can ONLY be locked AFTER a report is updated.**

PROF

The DeFindex vault requires:

1. **`get_manager()`** must be called to get the authorized manager address
2. **`report()`** must be called first to update strategy performance
3. **`lock_fees()`** must be called immediately after to calculate the fees
4. Both operations must happen **atomically** in a single simulation
5. **Manager authorization** is required to simulate fee locking

Without the router, you'd need multiple separate simulations, risking inconsistent calculations between calls.

## Contract Details

- **Vault Contract:** `CD4JGS6BB5NZVSNKRNI43GUC6E3OBYLCLBQZJVTZLDVHQ5KDAOHVOIQF`
- **Manager Account:** Retrieved dynamically via `get_manager()`

- **Network:** Stellar Mainnet

## Installation

```
npm install
```

## Usage

Run the script:

```
npm start
```

or

```
npm run dev
```

Both commands execute `src/report-and-lock-fees.mjs`

## What It Does

1. **Initializes** the Stellar Router SDK
2. **Simulation 1: Get Manager Address**
   - Calls `get_manager()` to retrieve the vault manager dynamically
3. **Simulation 2: Atomic Operations**
   - Creates two invocations:
     - `report()` - Updates strategy performance data
     - `lock_fees(None)` - Locks fees based on updated report
   - Executes atomically with manager authorization:

     ```
     const results = await sdk.simResult(invocations, {
       caller: MANAGER_ACCOUNT,
       source: MANAGER_ACCOUNT
     });
     ```

4. **Returns results** from both functions in a single transaction

PROF

## Total Simulations

Only **2 simulations** are executed:

1. **Simulation 1:** `get_manager()` - Retrieve manager address
2. **Simulation 2:** `report()` + `lock_fees()` - Atomic execution

# Use Case: Understanding Real Vault Value

## The Problem

When you call `total_managed_funds()`, it returns:

```
Total Managed Funds = Strategy Balances + Locked Fees + Idle Assets
```

But **locked fees are not available** for deposits/withdrawals - they belong to the protocol/managers. You need to know:

- **How much is actually available** for users?
- **How many fees will be deducted** on the next interaction?

## The Solution

By simulating `report()` + `lock_fees()`, you can calculate:

```
Available Funds = total_managed_funds() - pending_locked_fees
```

This is critical for:

- 🎯 **Frontend UIs** - Show accurate available balances
- 📊 **Analytics** - Calculate real TVL (Total Value Locked)
- 💰 **Trading Bots** - Make informed deposit/withdrawal decisions
- 🔍 **Auditing** - Verify fee calculations before transactions

# Sample Output

PROF

```
🔍 SIMULATION 1: Getting vault manager address...
✅ Manager retrieved:
GDQ4HYM5GRYMZX754BWXLCCE5UKNWVVLS2OEP5WNG6NXR6N4NWXA6QYA

🔍 SIMULATION 2: Atomic operations with manager authorization...

📊 Result 1: report() (Strategy Performance)
  Strategy 1: +115,721 tokens profit 📈
  Strategy 2: +127,270 tokens profit 📈
  Total Gains: +242,991 tokens

🔒 Result 2: lock_fees(None) (Pending Fee Calculation)
  Strategy 1: 23,144 tokens will be locked 🔒
  Strategy 2: 25,454 tokens will be locked 🔒

  💡 Total Fees That Will Be Locked: 48,598 tokens
```

```
✅ Simulation complete - NO on-chain state modified
✅ You now know exactly how many fees will be deducted on next
interaction
```

## Key Requirements

### 1. Dynamic Manager Retrieval

The script **dynamically retrieves** the manager address from the vault:

- Calls `get_manager()` at runtime
- No need to hardcode the manager address
- Ensures you always have the current manager

### 2. Atomic Execution

The router ensures both functions execute together:

- If one fails, both are rolled back
- Guaranteed consistent state
- Single transaction fee

### 3. Order Matters

The sequence is critical:

1. `get_manager()` retrieves the authorized manager
2. `report()` updates the strategy data
3. `lock_fees()` uses the updated data to calculate fees
4. Cannot lock fees without an updated report

## Implementation Pattern

PROF

```
// Simulation 1: Get Manager
const getManagerInvocation = [
  new InvocationV0({
    contract: VAULT_CONTRACT_ID,
    method: 'get_manager',
    args: []
  })
];
const managerResult = await sdk.simResult(getManagerInvocation);
const MANAGER_ACCOUNT = managerResult[0];

// Simulation 2: Atomic Operations with Manager Auth
const invocations = [
  new InvocationV0({
    contract: VAULT_CONTRACT_ID,
    method: 'report',
```

```
    args: []
  }),
  new InvocationV0({
    contract: VAULT_CONTRACT_ID,
    method: 'lock_fees',
    args: [StellarSdk.xdr.ScVal.scvVoid()]
  })
];
const results = await sdk.simResult(invocations, {
  caller: MANAGER_ACCOUNT,
  source: MANAGER_ACCOUNT
});
```

## Technical Details

- **Router SDK:** @creit-tech/stellar-router-sdk
- **Invocation Type:** `InvocationV0` (atomic batch execution)
- **RPC Endpoint:** `https://soroban-rpc.creit.tech`
- **Contract Functions:**
  - `get_manager()` - Returns the vault manager address
  - `report()` - Returns `Vec<Report>` with strategy performance
  - `lock_fees(new_fee_bps: Option<u32>)` - Simulates fee locking, returns fee reports

## Important Notes

### ⚠ This is a Simulation, Not Execution

- **NO on-chain state is modified** - All operations are simulated via RPC
- **NO gas fees are charged** - Simulations are free read operations
- **NO actual fee locking occurs** - This only calculates what WOULD happen
- **Results are predictive** - Shows fees that will be locked on next real interaction

PROF

### Workflow in Practice

```
1. Call this simulation → Get pending_locked_fees (e.g., 48,598 tokens)
2. Call total_managed_funds() → Get total funds (e.g., 10,000,000
tokens)
3. Calculate: available_funds = 10,000,000 - 48,598 = 9,951,402 tokens
4. Now you know the REAL available balance for users
```

This technique is essential for:

- Building accurate user interfaces
- Making informed investment decisions
- Calculating real protocol metrics
- Preventing transaction failures due to insufficient funds

## Dependencies

- `@stellar/stellar-sdk` - Stellar SDK for transaction building
- `@creit-tech/stellar-router-sdk` - Router SDK for atomic contract calls

## License

MIT