NES-UM-USA

RPG ELITE

INSTRUCTION BOOKLET

Printed in Japan

First Edition: May 2023

ISBN: 978-0-WeLoveThe80s-1

For a good time, check out more Soroban and its sweet documentation at our website, https://soroban.stellar.org

This may be a partial booklet, to view all available skirmishes visit https://rpciege.com/booklet/kit-and-caboodle

WELCOME TO THE WORLD OF RPCIEGE!

In an age where heroism is a currency, and villains lurk in the vast shadows of towering castles, exists the awe-inspiring realm of RPCiege. Stretching across diverse territories and brimming with secrets, this world will challenge every strand of your tactical prowess, daring you to rise, conquer, and rule!

Born from the ashes of the great conflict, a time of chaos, our world is scattered into numerous territories. Each unique in culture, resources, and strategic value. These lands are not void of leadership but are dominated by Overlords - ruthless individuals with an insatiable thirst for power.

But fear not, brave player, for you are the beacon of hope in this tumultuous world. You are one of the legendary heroes, entrusted with the noble mission to challenge these Overlords, reclaim the territories, and restore peace to the realm of RPCiege.

In your hands, you hold a deck of cards, each possessing unique abilities and power scores. These are your armies, your fortresses, your spies, and your diplomats. Use them wisely, strategize your attacks, fortify your defenses, form alliances, or break them when necessary. The key to victory lies in the balance of power and cunning.

But beware! The Overlords will not cede their territories easily. They too hold their own decks, filled with terrifying creatures and devious traps. Each turn, each move will lead you either one step closer to your destiny or into the jaws of defeat.

In the world of RPCiege, fortune favors the bold, and the cunning are crowned as kings and queens. So, delve deep, take command, and let your legend echo throughout the realm!

Welcome, brave adventurer, to your destiny. Welcome, to RPCiege!

Remember: In RPCiege, your mind is the ultimate weapon. Play wisely!

SHH. KEEP IT DOWN!

We're glad you're here, we desperately need your help. We're currently engaged in a siege of the Soroban RPC. The RPCiege! Contained within this booklet are instructions for tackling many terrific and terrible skirmishes. There are only three critically important pieces of information you'll need to remember.

1. Always use our official RPC endpoint when enacting your attacks https://testnet.rpciege.com along with the TESTNET network passphrase Test SDF Network ; September 2015

2. In order to issue digital collectible card awards we will observe contract invocations for a funded MAINNET Stellar public key at the final argument slot of the contract call. For game contracts we provide this will be an _nft_dest argument. For contracts you develop don't forget to include an Address as the contract's final argument

3. You can claim your NFT cards wherever you wish however we've built a simple claim page over at rpciege.com/claim to aid in the claiming process

BEWARE

Cards will be issued as rare animated assets for three weeks after the release of each Skirmish at which point they will begin being issued as common static assets.

Getting Started with Soroban

Before you begin your journey, you need only take your first step. If you're new here - these tips will be your guidestones.

Setup

Install and configure Rust and the Soroban CLI.

Hello World

Create your first Soroban contract.

Soroban RPC

The RPC service allows you to communicate directly with Soroban via a JSON RPC interface.

Soroban CLI

Soroban CLI is the command line interface to Soroban.

High-Level Overview

Descriptions of key Soroban Concepts.

Deploy to Testnet

Deploy and invoke your contracts on the Testnet network.

If you're prepared, continue on. **TO BATTLE!**

Skirmish I

For battle 1 all you've got to do is submit a contract invocation. That's it. Remember it needs to include as its final argument an Address of your mainnet Stellar public key where you'd like to receive your card pack for today's skirmish, but that's the only requirement.

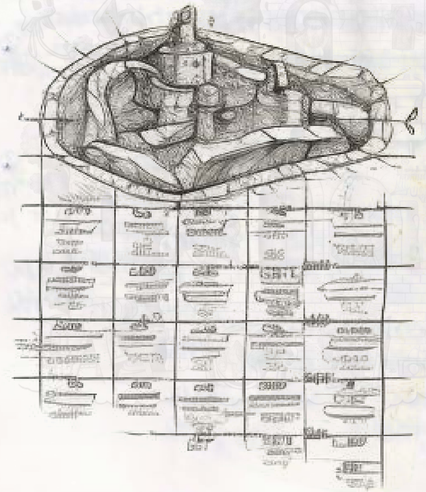
Here's an example contract, just because I like you ❤️

```
#![no_std]

use soroban_sdk::{contract, contractimpl, Address, Env};

#[contract]
pub struct Contract;

#[contractimpl]
impl Contract {
    pub fn run(_env: Env, _nft_dest: Address) {}
}
```

Skirmish II

Welcome to day 2 of the legendary RPCiege! For today's skirmish, your goal is to create and invoke a contract that returns the String 1694-1727.

Don't forget the three rules but other than that good luck and go nuts!



No contract cheat codes today but here's a link to some documentation you might find useful:

String in soroban_sdk - Rust

String is a contiguous growable array type containing `u8`s.

https://docs.rs/soroban-sdk/latest/soroban_sdk/struct.String.html

Skirmish III

Morning dawns on day 3 of the RPCiege. For today's quarrel we're enacting a new tactic: brute force. To win the battle you must submit an invocation to the `game_3` function of the CDUZZ624GWOCKRWO2POLWHRNM2YKOTBH7D6MXQIRDXQICJ46BXNM6JBQ contract.

Here's that contract's Rust code.

```rust
#![no_std]

use soroban_sdk::{
    contract, contracterror, contractimpl, panic_with_error,
    xdr::ToXdr, Address, Env, Symbol,
};

#[contract]
pub struct Contract;

#[contracterror]
#[derive(Copy, Clone, Debug, Eq, PartialEq, PartialOrd, Ord)]
#[repr(u32)]
pub enum Error {
    MissingPew = 1,
    UsedPew = 2,
}

#[contractimpl]
impl Contract {
    pub fn run(
        env: Env,
        symbol: Symbol,
        _nft_dest: Option<Address>
    ) -> Result<(), Error> {
        if env.storage().persistent().has(&symbol) {
            panic_with_error!(env, Error::UsedPew);
        }

        let bytes = symbol.clone().to_xdr(&env);
        let hash = env.crypto().sha256(&bytes);

        let mut i = 0;
        let mut has_pew = false;

        for v in hash.clone().iter() {
            if v == 112
                && hash.get(i + 1).unwrap_or(0) == 101
                && hash.get(i + 2).unwrap_or(0) == 119
            {
                has_pew = true;
            }
            i += 1;
        }

        if !has_pew {
            panic_with_error!(env, Error::MissingPew);
        } else {
            env.storage().persistent().set(&symbol, &true);
        }

        Ok(())
    }
}
```

No spoilers but I'm telling you now that writing a test against this contract will be your best ally.

Skirmish IV

A moment of silence for the comrades we've lost along the way😔. And onward into day 4 of the RPCiege! Today we employ the age-old tactic of meddling with the city's supply chain. We need to disrupt their storage and events. For that, we'll be calling the set and get functions on the CDOX4TLMDKYURVH7T3O5YWACYKPPMZ5HHC6RNUXHGWTSZ7APFZTHP6J4 contract.

I tried to acquire the contract code itself but was unable to. All I got was this instructional slip from inside an otherwise relatively bland fortune cookie.

```
env.storage().temporary().set(&bytes, &rand_array_store);
env.events().publish((bytes.clone(),), rand_array_event);

# soroban contract invoke --id <contract-id> --source <saved-identity> --network <saved-network> -- <?function-name> -h

# https://github.com/stellar/soroban-tools/blob/main/docs/soroban-cli-full-docs.md#soroban-events
# https://soroban.stellar.org/api/methods/getLedgerEntries
```

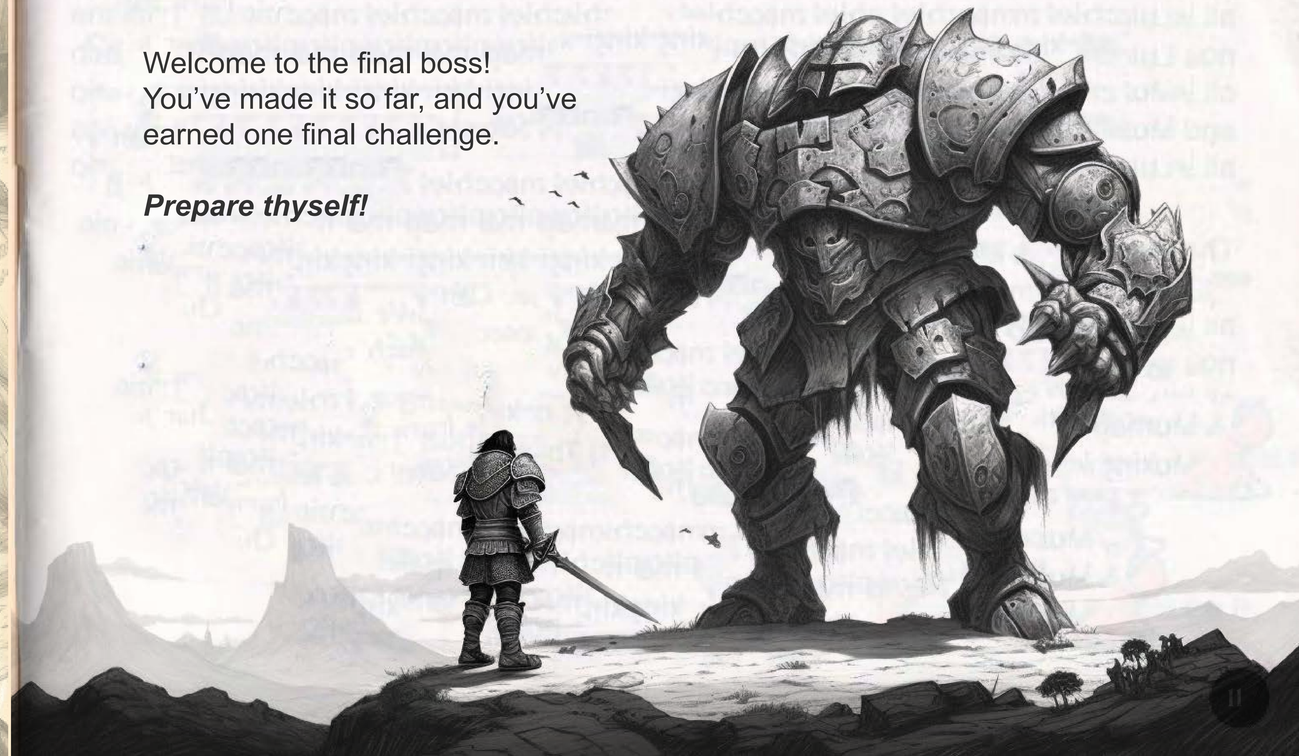
It would appear you'll be on your own for this one.

GOOD LUCK...

Skirmish V

Welcome to the final boss!
You've made it so far, and you've earned one final challenge.

Prepare thyself!



TL;DR

Today's first task is to deploy a new Liquidity Pool using our *deployer* contract `CBNTWL5RFKILVFLBNCF7EYHAN7RTHWA3ZXHWWLTLHNX5E6HBY3DBQHCI`. This contract will emit an *event* containing more "instructions."

Good luck!

The Task at Hand

1. Issue Two New Assets
2. Wrap the Assets for Use as Soroban Tokens
3. Deploy New Liquidity Pool Contract
4. Deposit into the LP
5. Find Your Deposited Tokens
6. Learn About Callback Functions
7. The Problem in this Instance
8. Wait... What is Actually Happening Here!???
9. Now, Go Get Your Tokens

Issue Two New Assets

You'll need to brush off your "regular Stellar" hats for this part. We'll wait while you get ready (but the other players probably won't, so don't dawdle).

You'll need (at least) two accounts for this (any randomly generated addresses will do, and we'll refer to them as the `rpciege05` account and the `issuer` account). There are a few different ways you could go about this step, but here it is quick and simple:

Note: For the rest of this document, we may reference these two assets with the codes `assetA` and/or `assetB`. You can use those asset codes, or choose your own. When you read `assetA` or `assetB`, make sure you mentally think about which asset that might be for your use. Importantly, use your assets consistently as `assetA` or `assetB`, and don't swap them around.

1. Use the Stellar Laboratory to interact with the `testnet` network.

2. Use the **Create Account** page to generate two new keypairs, and fund both of them using Friendbot. Don't forget to copy/paste those public/secret keys somewhere for future use.

3. Use the **Build Transaction** page to create a transaction using these options:

- *Source Account*: use your `rpciege05` account for this
- *Sequence Number*: click the *fetch next sequence number...* button
- *Base Fee*: Normally, leaving this set to `100` is totally suitable for testnet operations. You are, of course, free to choose a higher base fee. (This can help ensure transaction success in times of high network volume.)
- *Operations*:
  - Add a `Change Trust` operation, with an *Asset Code* of your choosing, and the `issuer` public key for the *Issuer Account ID*
  - Duplicate this to a second `Change Trust` operation, changing the *Asset Code*, but keeping the same *Issuer Account ID*
  - Add a `Payment` operation, using the `rpciege05` public key for the *Destination*, either *Asset* from the previous operations, any *Amount* you like (greater than `1`, though, otherwise the stroop-maths get confused), and the `issuer` public key for the *Source Account*
  - Duplicate this to a second `Payment` operation, using the other *Asset* from the previous operations (you could change the *Amount*, too, if you want)
- Altogether, that makes a transaction that contains two `Change Trust` operations and two `Payment` operations for two unique assets issued by the `issuer` account.

4. Click the *Sign in Transaction Signer* button at the bottom of the page. You'll be taken to the **Sign Transaction** page, where you will need to copy/paste the secret keys of both the `rpciege05` and `issuer` keypairs.

5. Click the *Submit in Transaction Submitter* button at the bottom of the page. You'll be taken to the **Submit Transaction** page, where you can click the *Submit Transaction* button, and wait for the success confirmation.

Note: If your transaction is having trouble making it to the ledger (you might get a timeout error code `500`), you can try increasing the *Base Fee* of the transaction to help boost your chances of a successful transaction.

Congratulations! You've now minted two brand new assets into existence on the Stellar network!

Note: We breezed through this part, since asset issuance isn't really the primary point of today's skirmish. If you're interested in learning more, check out Level 01 of Stellar Quest for lots of quality payment- and asset-related content.

Wrap the Assets for Use as Soroban Tokens

Soroban makes it very easy to interact with previously issued Stellar assets. But, these assets *do* need to be “wrapped” first in order to make them available for use in Soroban. This process is quick and painless using the soroban-cli.

First, let’s set up the network, accounts, etc. we will use for the CLI. This isn’t strictly necessary, but will make *everything else* that much easier.

```
# create an `rpciege` network configuration that includes the network passphrase and rpc url
soroban config network add \
   --rpc-url https://testnet.rpciege.com \
   --network-passphrase ‘Test SDF Network ; September 2015’ \
   rpciege

# create an `rpciege05` identity configuration from the corresponding secret key
soroban config identity add \
   --secret-key \
   rpciege05
# paste in the `rpciege05` secret key when prompted

# create an `issuer` identity configuration from the corresponding secret key
soroban config identity add \
   --secret-key \
   issuer
# paste in the `issuer` secret key when prompted
```


Now that we've got our accounts and network in place, we can actually get to work! Hooray! You'll need to run the following command *twice*, once to wrap each of the assets you trusted and paid in the transaction earlier.

```
soroban contract asset deploy \
  --network rpciege \
  --source rpciege05 \
  --asset <asset-code>:<asset-issuer-public-key>
```

In response to a successful invocation, you'll get a contract address of the wrapped asset that can be used in Soroban. These contracts will have the same interface and functions available as any token using the Stellar Asset Contract. You should copy/paste these contract addresses into a safe place for later use.

So, let’s take a breath and recap the progress we’ve made so far:

- You used your rpciege05 account to create two new trustlines for assets.
- You used your issuer account to make payments of those assets to your rpciege05 account.
- You used the soroban-cli to wrap those assets so they can be used as Soroban tokens.

Now, you’re ready to actually deploy a liquidity pool!

Deploy New Liquidity Pool Contract

When someone builds a liquidity pool project (Uniswap, Sushi, etc.), they want all the liquidity pools to follow the same logic. So, instead of writing a unique contract foreach and every asset pair (as trivial as that task might be), a "vanilla" LP contract's bytecode can be installed on the network. This bytecode will then be duplicated and initialized as a new contract for each new asset pair LP.

Our *deployer* contract (sometimes called a "factory") will take care of all that work foryou! (You're welcome.) When you invoke the `new_liqpool` function of our *deployer* contract, the *deployer* contract:

1. Ensures the deployer contract has been initialized. (We've taken care of this step already, so you don't have to worry about it. But, the deployer contract still does make that check.)

2. Uses Soroban's `Env::deployer()` function to (you guessed it) deploy the pre-installed WASM bytecode to a new contract (using the salt you provided atinvocation-time to generate the new pool's contract address).

3. Sets up a client for the newly deployed LP contract, and runs the `initialize` function using the token contract addresses you provide at invocation-time.

4. Uses Soroban's `Env::storage()`utilities to store the contract address of the newly deployed LP (this helps us to check your work, don't stress about it too much).

5. Sends back to the invoker the `contract_address` for their new liquidity pool, ready to receive deposits and be used by users.

To make this happen, first we need some salt (the spice of life)! Soroban contract addresses are (at least partly) determined by the address of the contract's deployer. Since we are deploying many LP contracts from the same deployer contract, we'll want to make sure there is some unique entropy to generate each contract adress from. You'll need to supply a 32-byte (64-characters) hex string as part of the `new_liqpool` invocation. If your system has `openssl` installed, you can use the following command to generate something useable:

```
openssl rand -hex 32
```

Alternatively, you could hand-write your own, and see what fun words you might be able to create from strictly hex characters. (I personally like `deadbeef` !)

In any case, once you have a salt of some kind, you are ready to get into the pool! You'll need to know the `contract_address` for the two assets you previously issued.

Note: It's been a while since we've mentioned it, so it's worth repeating: `CBNTWL5RFKILVFLBNCF7EYHAN7RTHWA3ZXHWWLTLHNX5E6HBY3DBQHCI` is the `contract_id` of our deployer contract. This is the invocation where it will do all the things we described a few sentences ago (check for initialization, deploy a new instance of the pre-installed bytecode, etc.).

```
soroban contract invoke \
    --network rpciege \
    --source rpciege05 \
    --id <lp-contract-id> \
    -- \
    new_liqpool \
    --salt <your-previously-generated-salt-hex> \
    --token_a <assetA-contract-address> \
    --token_b <assetB-contract-address>
```

Copy/paste the returned `contract_address` for use in later steps.

Deposit into the LP

Amazing! You are the proud creator of a brand new liquidity pool! That wasn't even that hard, right!? Now, we're ready to actually make a deposit! You'll want to decide on the amount of each asset you'd like to deposit (and know how to convert that amount into). (The short and simple explanation for stroops is that it's the "decimal precision" of an asset on the Stellar network. In this case, multiply the amount you'd like to deposit by 10,000,000 to get the number of stroops you should deposit.)

You can use the tools you've learned previously to figure out the proper way to invoke a function on this LP contract. Here's how you can find the `--help` output for it:

```
soroban contract invoke \
    --network rpciege \
    --source rpciege05 \
    --id <lp-contract-id> \
    -- \
    --help
```

This will show you all the available functions on the contract, and you can add a function name after the `--` double-dash (it's called a slop, btw) to get explanations for each required argument:

```
soroban contract invoke \
--network rpciege \
--source rpciege05 \
--id <lp-contract-id> \
-- \
swap \
--help
```

From the looks of it, the `deposit_liquidity` function is most likely what we are after. Let's invoke it and make a deposit of our two issued assets. Below, is an example depositing `100` units of `assetA` and `200` units of `assetB` (I've already multiplied to arrive at `1,000,000,000` and `2,000,000,000` stroops of each asset, respectively).

Since we are the *very first* depositors into this liquidity pool, we get to determine the relative value of each asset's reserves. We are depositing using a `1:2` ratio of `assetA` to `assetB`, but you can deposit however much you like in yours. I've used `0` as the minimum arguments since there is no possibility of slippage (i.e., there are no reserves that need to be calculated against). In a real-world LP that already contains reserves of the pool assets, you would want to utilize some sensible minimum amounts to ensure your assets retain the correct relative value during the deposit.

```
soroban contract invoke \
--network rpciege \
--source rpciege05 \
--id <lp-contract-id> \
-- deposit_liquidity \
--addr <rpciege05-account-public-key> \
--desired_a 1000000000 \
--min_a 0 \
--desired_b 2000000000 \
--min_b 0
```

Well done! You've not only *created* a liquidity pool on Soroban, you've now also made an initial deposit into it! You're on fire! Now, how do we get those sweet yields paid back to us!?

Find Your Deposited Tokens

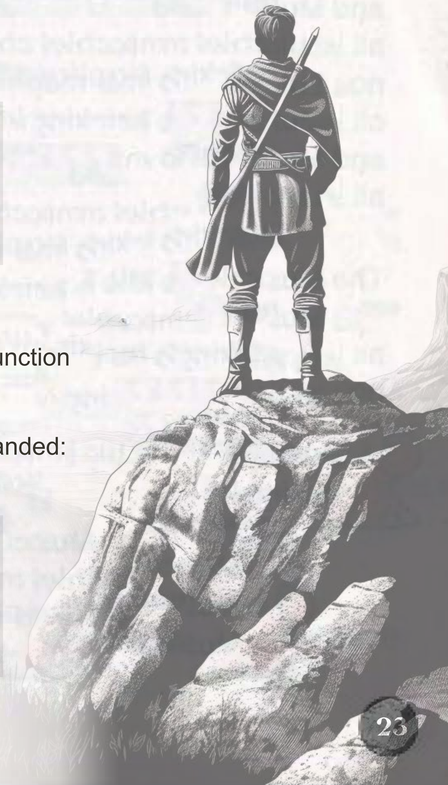
Let's see the help output of our contract again, and see if it says anything about how we can withdraw, or something like that:

```
soroban contract invoke \
--network rpciege \
--source rpciege05 \
--id <lp-contract-id> \
-- --help
```

That's really strange... The only functions available are `swap`, `get_reserves`, and `deposit_liquidity`. There's not even a function to make a withdrawal!? I'm starting to get nervous!

Let's at least double-check the reserves, to make sure our deposit landed:

```
soroban contract invoke \
--network rpciege \
--source rpciege05 \
--id <lp-contract-id> \
-- get_reserves
```



The output shows us that the LP holds `["1000000000","2000000000"]` reserves of `assetA` and `assetB`, respectively. So, at least the tokens aren't lost, right! They're in the pool, where they "should" be?! (I'm not convincing myself to be calm, either, by the way.)

There is one more way to find some information about a given Soroban transaction: events! Many transfers, deposits, and other functions programmed for Soroban will emit some type of `event` that can be seen in the transaction metadata. Maybe our LP emits those events for deposits. Let's cross our fingers, and head to command line!

Note: Events are like a smart contract "secret sauce" that can provide a ton of information and insight into what's happening with a contract. Once you're done with this skirmish, you should **definitely** go read up on them some more!

You *could* get an event output from a Soroban invocation by passing `--events` in front of the `--` double-dash in your `deposit_liquidity` command. We understand, though, if you're a bit hesitant to put even more of your hard-earned tokens at risk just in the hopes of investigating further. Not to worry, you can also get events from the soroban-cli, no contract invocation required.

```
soroban events \
--network rpciege \
--id <lp-contract-id> \
--start-ledger <ledger-number>
```

Aside from the contract ID, the only piece of information is what ledger number to start your search from. You can get some key information from the RPC server, to help you make an educated guess about what to use for this argument. We want to invoke the `getLatestLedger` method of the RPC server. Soroban-RPC will accept HTTP POST requests using the JSON-RPC 2.0 specification. You can make these requests using curl, Postman, httpie, or whatever means you are comfortable with. The essence of what you want to do is send a JSON object to the server, and look through the response for the latest ledger number. The object you want to send should (for the `getLatestLedger` method) include the fields `jsonrpc`, `id`, and `method`:

```
{
  "jsonrpc": "2.0", // this always has to be here, and should always b e `2.0`
  "id": 0, // you can use any integer here
  "method": "getLatestLedger" // this is the method we want executed
}
```

Here's how you might make the request using curl:

```
curl -X POST -H 'Content-Type: application/json' \
-d '{"jsonrpc":"2.0","id":0,"method":"getLatestLedger"}' \
https://soroban-testnet.stellar.org
```

An unofficial Soroban-RPC Postman collection is also available for use, and contains boilerplate for the existing Soroban-RPC methods.

The returned object will contain a `sequence` key, and the associated value is the sequence number of the most recently closed ledger on Tesetnet. If you've just invoked the `deposit_liquidity` function of your LP, you should be able to just subtract a few ledgers from this number and successfully get information about your deposit event.

If it's been some time since your LP invocation, you could look up your account at the horizon endpoint for accounts, and use the `last_modified_ledger` field to make an educated guess about what to use for this argument.

If it's been more than 24 hours since you deposited into your LP, the RPC server will likely not have any record of that event. You'll need to look to Horizon instead, and find the transaction's `result_meta_xdr` which contains the event output, as well. There are so many ways to get details about events!

Back to the contract events we're looking for, you should see a `CONTRACT` event's output that contains the topics `deposit` and your `rpciege05` public key (it's in hex, instead of the more common `G...` address format. But it's the same key, trust us). Awesome! We're on the right track.

This event has a pretty *hefty* amount of bytes associated with it... I wonder what's contained there??! This can be a bit tricky to get meaningful text out of. But, you can accomplish what you need in node.js pretty easily. is a great option for this!

```
let eventMessageBytes = Buffer.from('<hex-encoded-bytes>', 'hex');
console.log(eventMessageBytes.toString('utf-8'));
```

Reading the output... Oh no!!! We got scammed! Fortunately, someone on the development team of this liquidity pool has given us some clues as to how we can get our tokens back! Thank you, kind sir or madam!

Before we get to that; a bit of theory (don't worry, we'll try to keep it brief).

Learn About Callback Functions

A common convention in smart contract development is the use of "callback functions." These are functions passed by the invoker of the contract, which will be run as part of the atomic transaction (that just means the "callback function" will be executed inside the initially invoked function, and everything either fails or succeeds together).

In the Uniswap protocol for example, an app invoking the `swap` function on a liquidity pool must provide its own `swap_callback` function that will be run in the course of each `swap` invocation it (the app) makes. Inside this callback function the app might get authorization from the user, receive payment from the user, ask the user about their favorite color, spend the user's gas money on chocolate milk, or just about anything else. This function is a so-called "callback" because it is provided as a parameter at the time of invocation. Uniswap doesn't write the function, the app developer does.

The expectation of the Uniswap lp protocol is that the callback function can do pretty much whatever it wants, *as long as the lp gets paid properly*. For a successful swap to take place, the lp **must** get paid (at least) the amount it expects of the counter asset, before it will send any of the purchased asset anywhere.

The Problem in this Instance

These callbacks can be very powerful and useful tools, but they can also be made to operate in unexpected ways if the original LP protocol (Uniswap v3 in this case) doesn't do the proper checks. The contract we've written (the one that has scammed you) is vulnerable to the same type of exploit that (nearly) hit Uniswap v3.

Note: The bug in Uniswap's code was detected by an audit before v3 was in production. This does make for an interesting case study, but no real user funds were ever actually in danger of this vulnerability.

The liquidity pool contract we've written has a `swap` function which is expecting to be invoked with a `callback` argument: that is, the address of a contract that implements a `swap_callback` function. This callback can do whatever it likes, as long as our liquidity pool is paid the proper amount of the counter asset by the time the callback has finished.

Here's what this callback invocation looks like in our liquidity pool contract:

```
// call the `swap_callback` function on the callback contract
// For the swap to be successful the callback should send the specified
// `amount` of `token_in ` to the liquidity pool
let callback_client = ::Client::new (&env, &callback);
callback_client.swap_callback(&env.current_contract _address(),
  &token_in, &amount, &from
);
```

After the `swap_callback` function successfully completes, we then run a quick check to make sure our maths are all working out correctly:

Hint: This check is where the problem is...

```
// we check that at least the correct amount was sent
// to the liquidity pool
// if not we return the liquidity pool's `InNotSent` error.
if in_before + amount < token_in_client.balance(&env.current_contract_address())
{
   return Err(Error::InNotSent);
}
```

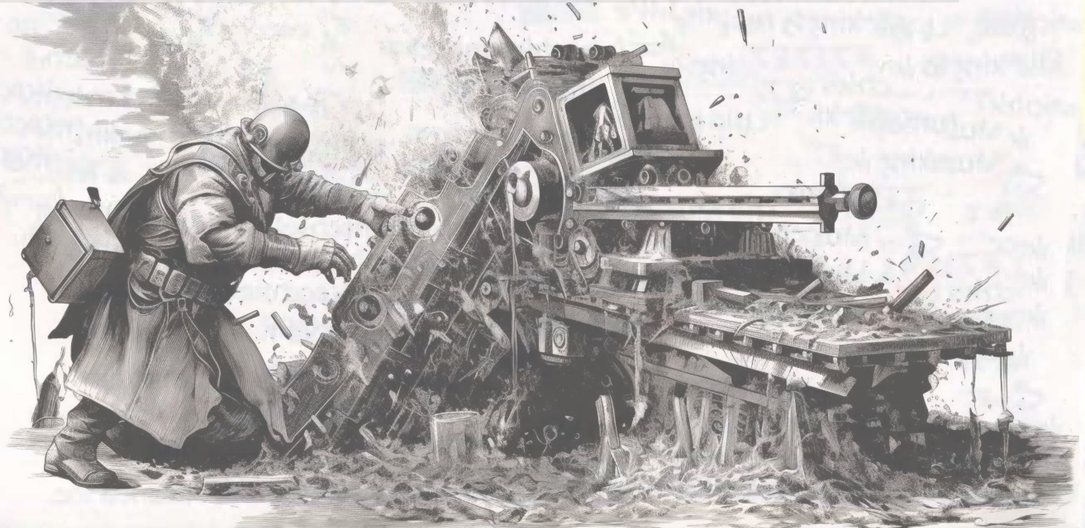
It's easy to miss why this is vulnerable, but it all rests in how the maths are compared. For the sake of simplicity, let's imagine an LP with a `1:1` ratio of reserves (we'll also ignore fees/yield for simplicity). The current reserves are `["500", "500"]`, and you'd like to sell `5` of `assetB` to buy `5` of `assetA`. So, the check would end up like this:

```
// the amount of `assetB` in reserves before the swap
let in_before = 500

// the amount of `assetB` you are "selling" into the pool
let amount = 5

// this is performed _after_ the callback function has completed, so it has the most up-to-date
// balance of the amount of `asssetB` held by this LP contract. In theory this _should_ be 505
// (the callback sent 5 `assetB`), but let's exploit the bug, and send only **1* * of `assetB`,
// so it's _really_ 501!
let current_balance = token_in_client.balance(&env.current_contract_address())

// the comparison goes like this, if we substitute the actual values:
if 500 + 5 < 501 {
   // throw an error
}
```

Note: For any curious minds out there, we don't check for equality == because the callback may have sent more than we expected. If that's the case, we don't really care - just more liquidity to swim in!

Since the maths check out, we could even send nothing to the liquidity pools, and stillreceive the amount of asset we're trying to purchase! The problem is simple: areversed comparison operator. The fix is equally simple and we would write thecomparison like this:

```
// the fix would be to reverse the operator, so
// `if in_before + amount > token_in_client.balance() { Error }`
// This would assert that no less than amount was deposited.
if in_before + amount > token_in_client.balance(&env.current_contract_address())
{
    return Err(Error::InNotSent);
}
```



Wait... What is Actually Happening Here!???

I know that's a lot of text, so here's the bullet-points summary of what's happeningduring this swap:

- A user invokes the `swap` function on the liquidity pool contract, providing the address of the `callback` contract as an argument
- The LP contract takes stock of the following items:
    - Which asset is being sent *into* the LP
    - Which asset is being taken *out of* the LP
    - The amounts of each asset on each side of the LP
    - How much in fees (and in which asset) are expected for the swap
- The LP contract then invokes the `swap_callback` function of the callbackcontract
- the callback contract does whatever it's programmed to do in the `swap_callback` function
    - As part of the `swap_callback` function, the callback contract is supposed to send the correct amount of the "input" asset to the LP contract
- The LP contract does (incorrect, in this case) maths to make sure the paymenthas been received from the callback contract
- If the maths work out (i.e., the payment from the callback function was"successful"), the LP contract will send the amount of the "purchased" asset to the callback contract
- The LP contract then updates its reserves storage entries to reflect the new balances of the assets

Now, Go Get Your Tokens

Alright, enough theory, you have tokens to retrieve! You'll need to write and deploy a callback contract that implements a `swap_callback` function. That function can do whatever you want, but in order to receive your tokens back (and win this skirmish) you want it to **NOT** send any tokens of either asset to the LP, thereby exploit the bug and "hack yourself" into getting your own funds back from the malicious LP.

Following Uniswap's pattern, the tokens from our LP are ultimately sent from the LP contract's balance to the callback contract's balance. This makes it possible for the callback contract to be a protocol and/or wallet, in which case the user's funds are kept in that same wallet following the swap. In order for this exploit to benefit a specific user, that user would need to deploy their own callback contract, and include some mechanism for the funds to be transferred from the callback contract to the user's account.

Note: Transferring tokens from the callback contract back into your `rpciege05` account is not required to complete this RPCiege skirmish. As long as you can manage to exploit the vulnerability, RPCiege doesn't care where your funds ultimately land. Even if it's in a burner callback contract.

Here's something to get you started, but it's not complete:

```
#![no_std]

// HAHAHAHAHA YOU THOUGHT!!!
// THIS IS SKIRMISH 5 YOU SILLY GOOSE!
// GET BUSY!

mod test;
```

Once your callback contract is compiled and deployed, you must determine the correct way to invoke the `swap` function on your liquidity pool. Your goal is to make a swap that returns deposited tokens to you, without having to pay further into the LP. Good luck!

One more thing!! When you're ready to invoke the `swap` function, don't forget to include the `_nft_dest` argument so you can claim the prize for which you've fought!

Midnight Madness

GAME EXPANSION

Skirmish VI

Following the downfall of that big bad boss, you probably basked in the glow of victory. Merriment echoed in every corner, songs of triumph whispered by the wind. Yet, as they say, when one evil falls, another often rises to take its place. Or in our case, a horde of them! With the veil of night, our once serene lands have taken on a decidedly sinister aspect. Monsters of yesteryears' tales, now very much real, have decided to come out and play. Their playground? Our home!

Creeping ghouls, vampiric high society, moon-howling werewolves, and cackling witches, all with an unfortunate taste for mayhem. Our everyday citizen would usually run, but surely not you.

We need to call contract `CAKR5PX3O2VUN3MFRHZRZSZYAPERPXVLOB6EATB7LYJDPYGU6D7IFDKV` to `set_traps` preemptively so that when nightfalls we can capture the beasts even as they rise from their pits of murky darkness.

Good luck! Don't

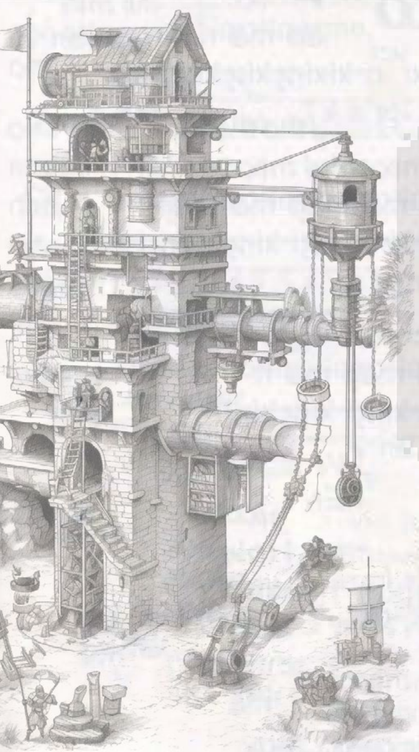
```
Error::YouDied
```

```rust
#![no_std]
#![feature(iter_array_chunks)]

use rand::rngs::SmallRng;
use rand::{Rng, SeedableRng};

use soroban_sdk::xdr::ToXdr;
use soroban_sdk::{
    contract, contracterror, contractimpl, contracttype, panic_with_error, Address, Env, IntoVal,
};

const WEEKS_IN_LEDGERS: u32 = 151200 * 4;

#[contracterror]
#[derive(Copy, Clone, Debug, PartialEq)]
#[repr(u32)]
pub enum Error {
    TrapNotSet = 1,
    YouDied = 2,
}

#[contracttype]
pub enum DataKey {
    TrapXY(Address),
}

#[contract]
pub struct Contract;

#[contractimpl]
impl Contract {
    pub fn set_trap(env: Env, x: u32, y: u32, source: Address) -> Result<(), Error> {}
    pub fn nightfall(env: Env, source: Address, _nft_dest: Option<Address>) -> Result<(), Error> {}
}

fn get_entropy(env: &Env, source: &Address) -> u64 {}
```

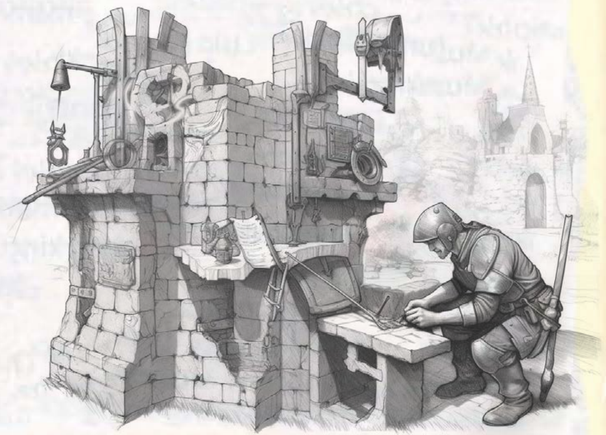


```rust
pub fn set_trap(env: Env, x: u32, y: u32, source: Address) -> Result<(), Error> {
    source.require_auth_for_args((&source,).into_val(&env));

    let trap_xy = (x, y);

    env.storage().temporary().set(&DataKey::TrapXY(source.clone()), &trap_xy);
    env.storage().temporary().bump(&DataKey::TrapXY(source), WEEKS_IN_LEDGERS);

    Ok(())
}
```

```rust
pub fn nightfall(env: Env, source: Address, _nft_dest: Option<Address>) -> Result<(), Error> {
    source.require_auth();

    let mut rng = SmallRng::seed_from_u64(get_entropy(&env, &source));

    let monster_xy = (rng.gen(), rng.gen());

    let trap_xy = env
        .storage()
        .temporary()
        .get::<DataKey, (u32, u32)>(&DataKey::TrapXY(source))
        .unwrap_or_else(|| panic_with_error!(&env, Error::TrapNotSet));

    if monster_xy != trap_xy {
        panic_with_error!(&env, Error::YouDied);
    }

    Ok(())
}
```



```rust
fn get_entropy(env: &Env, source: &Address) -> u64 {
    let sequence = u64::from(env.ledger().sequence());
    let mut entropy: u64 = u64::MIN;

    for [a, b, c, d] in source.to_xdr(env).iter().array_chunks() {
        entropy = entropy.wrapping_add(u64::from_be_bytes([a, 0, b, 0, c, 0, d, 0]));
    }

    // modulo 6 tolerance to account for the time delay between simulation and submission
    entropy = entropy.wrapping_add(sequence - sequence % 6);

    entropy
}
```



Skirmish VII

Troubles never come singly, do they? As if our monstrous infestation wasn't enough, our wells have gone from providing water to distributing curses. The citizens are changing, morphing into beings of the undead! One moment, they're arguing about the weather, the next - they're craving for brains! We quickly need to develop a purification contract which can take in a random `Bytes` array of `u8` molecules and return it back ordered from smallest to largest and containing only pure prime values.

Once you've designed and deployed a contract containing a `purify` function call which meets the criteria above, pass it to the `run` function of the `CCP7F5PC7I3CIR6DNIFFNMSHPAZXMDRDR5UCYIUEYIIJTZ2CVM4DYR2V` contract to test your purifier contract. The fate of all men, women and children rest in your hands. Don't muck it up!

Skirmish VIII

Will this torment never end? The townspeople are very upset (and who could blame them really with all that spooky stuff happening). They cry for help from the kingdom's famous investigative duo, the Cipher and the Cryptographer. Though monsters and cursed wells have plagued the citizens, the Cipher and the Cryptographer have unearthed a deeper, more subtle menace. A secret code has been engraved into the very fabric of our world, manifesting as a sequence of unpredictable events. These events lead to chaos and disorder, sprouting distractions like monsters and cursed wells. Our reality is essentially a cipher waiting to be decoded!

The duo has put their collective knowledge to the test, writing tomes filled with enigmatic riddles and conjectures. They've embedded clues within a `ContractEvent`. Use them to gauge if you're onto the sequence. Only the most astute can detect the pattern, reveal the code, and thereby remove the chaos at its root.

To participate, you must call the contract CC75LGT3V5L6IZGWDJHFJHSJIMCGWKEMQ6DFPP2JFE7ZGWJ25PEVE7FI and try your luck in the guessing game. Should you guess incorrectly, observe the emitted event. Decipher it, for it holds the clues you need to unlock the pattern.

The Dawn Watchman, vigilant as always, will bear witness to your trials. He warns, however, that "errors lie not just in numbers but in their sequence." Take heed, and bring order to the chaos.

Skirmish IX

Amidst these challenges, a new invention has been crafted by the citizens—a magical contract deployer identified by CAQ6LJSUTMAXD3G2PAPLSEWRSTRBHWB4CQ6SBKBC75VF4GHTCA7CZ462, meant to protect the realm. However, its produce is incomplete; like slingshots without pockets.

Our Rabble Rouser, no stranger to agitation, has been haranguing the streets, demanding that this ineptitude be fixed.

"We've been given a lock without a key!" he shouts, drawing attention and challenging the community's builders to complete the contract.

You must develop a patched contract to use as an upgrade for the deployer's defective spawn by adding a missing argument that will make the contract complete and functional.

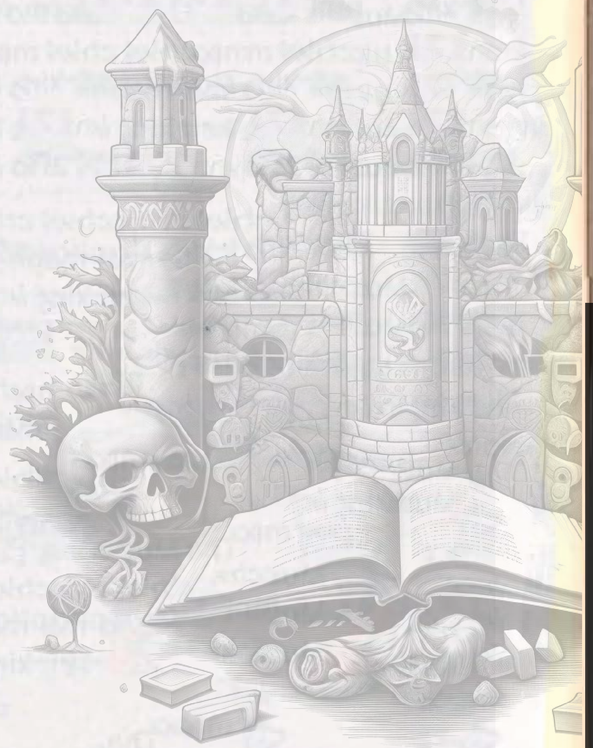
The Cryptographer winks at you, "Failing to upgrade is like trying to read a book missing its most crucial chapter. Don't leave the story half-told."

Skirmish X

OH NO!

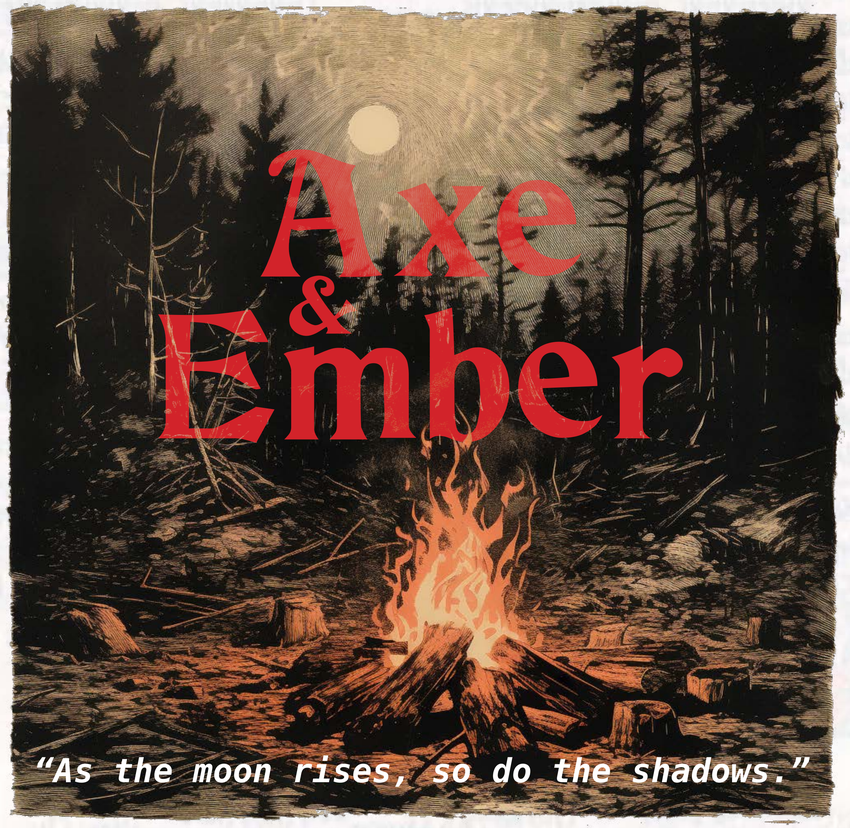
There's been a disturbance in the tapestry of fate! A so-called "Fund of Destiny" has been set up, but it's rigged in favor of a dark power that plans to use its winnings to usher in an era of torment. This fund relies on a future random number, and the Seer's crystal ball is cloudy with interference.

```rust
use rand::{rngs::SmallRng, Rng, SeedableRng};
use soroban_sdk::{contract, contractimpl, Address, Env, IntoVal};

use crate::types::{DataKey, Error};

#[contract]
#[allow(dead_code)]
pub struct Fund;

#[contractimpl]
#[allow(dead_code)]
impl Fund {
    /// Try winning the fund by providing your guess
    pub fn shuffle(
        env: Env,
        addr: Address,
        guess: u32,
        _nft_dest: Option<Address>,
    ) -> Result<(), Error> {
        addr.require_auth_for_args((&addr,).into_val(&env));

        let state = u64::MIN
          .wrapping_add(env.ledger().timestamp())
          .wrapping_add(env.ledger().sequence() as u64);
        let mut rng = SmallRng::seed_from_u64(state);

        if guess != rng.gen_range(0..1_000_000_000) {
            return Err(Error::WrongGuess);
        }

        // if the guess is correct, mint a new FundWon token
        env.storage().persistent().set::<DataKey, i128>(
            &DataKey::Balance(addr.clone()),
            &(env
                .storage()
                .persistent()
                .get(&DataKey::Balance(addr))
                .unwrap_or(0)
                + 1),
        );

        Ok(())
    }
}
```



Cue the Blind Hermit, his mind tangled with truths unspeakable, who speaks in riddles: “The wheel of chance is but a lie; future’s veil you must untie.”

You must predict this “unpredictable” sequence by sending a cross contract call to CAO7ZQB2LHXEFTM6IX63CDNUV4ERGXMVJYABXU37KQYDSYTPOVA2QM2H. A precise guess, made in the same ledger sequence, will crack the game wide open, diverting the ill-gotten gains and perhaps changing our destiny.



You’re a traveling rock peddler journeying to a new town.

Your path has unexpectedly led you into the depths of **DREADWOOD FOREST** on the night of the full moon. You’ve heard stories of these woods and the monsters that emerge when the sun goes down. And you know these dangers are even more menacing under the silvery gaze of the full moon’s light.

You are in a lush grass-covered clearing surrounded by dense pine trees when you decide to make camp for the night. You must survive until morning by keeping the monsters at bay with artificial light, or by trying something a little more creative. Monsters can attack anytime once the sun sets (from 9 p.m. to 5 a.m.).

In addition to the dull axe, this contract address CCCS6PRQC7AT6GC4BTY3H5CVPWSD57XM5J6ZAXYLMNA4QZ32YNDNB5LI, and some flint hanging from your belt, you have a knapsack with two items in it:

[1] Oil lamp and canister of oil

[2] Torch and sharpening stone

[3] Flask of whiskey and lucky amulet

https://www.youtube.com/playlist?list=PLmLehibUHqKMnGCiaRVmN5aCDkK7Xn_sH

Skirmish XI

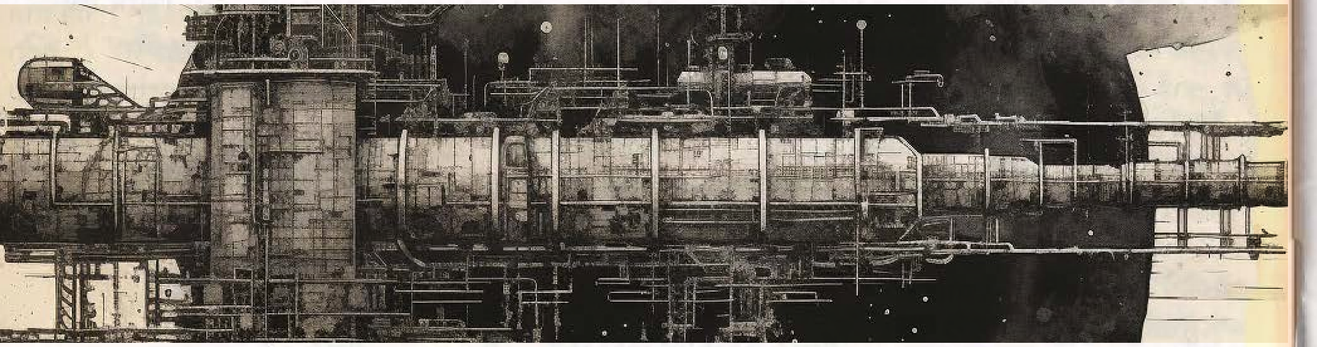
You've overthrown warlords, conquered monsters, and vanquished the deeper menace. You're exhausted (and rightfully so, that's a lot of stuff). But the battle isn't over yet, and to continue, you're going to need an ally. You've heard whispers from your fellow soldiers of a legendary figure whose battle prowess is only matched by her wisdom. They speak of the astute archaeologist called Blossom Bernice Breydenblach (Bee), and you think she can help. But there's only one problem. She's in space.

To retrieve her, you must travel to the cosmos yourself, battle celestial foes, and bring Bee back down to Earth.

First up, it's time to get your old spaceship out of storage and make sure it's got a valid registration. You head to the storage facility, where your beloved ship, Tantive V, has been gathering dust for the past fourteen years. You locate your storage unit and see that the registration paperwork is way out of date.

To successfully register your ship and get it out of storage, you must bump both an instance and persistent storage key from within a Soroban smart contract. Keep in mind that to accomplish this successfully the bumps *must* actually occur. **It will not be sufficient to merely include the bump calls in your contract.**

Skirmish XII



You've just spent five hours at the DSV (Division of Space Vehicles) renewing your registration and Tantive V is ready to go! You fire her up and blast off into space.

Entering into the planet Arda's orbit, you see three separate space stations titled `Station::Persistent`, `Station::Temporary`, and `Station::Instance` (don't look at me, I didn't name them). You need to store these space stations into your ships database under the appropriate storage types in order to run cross quantum analytics to decipher the station that contains Bee to continue on your quest to bring her Earth-side.

Build, deploy and invoke a contract storing these `Station::Persistent`, `Station::Temporary`, and `Station::Instance` keys inside a reasonable corresponding storage type.

Skirmish XIII

Now that you've successfully boarded the space station containing the illustrious Blossom Bernice Breydenblach, it's time to figure out where the heck she is. You slink through the burnished corridor, boots clicking smartly on the metal floor.

You're just about to round a corner when a large shadow steps into your path. Gasping, you stumble backward, gazing up at a tall alien with a crimson head and luminescent grey eyes.

"I come in peace!" you say, throwing your hands up in surrender.

The alien looks at you blankly before glancing at a clipboard and asking, "Are you the driver of spaceship Tantive V?"

"Y-y-yes," you answer.

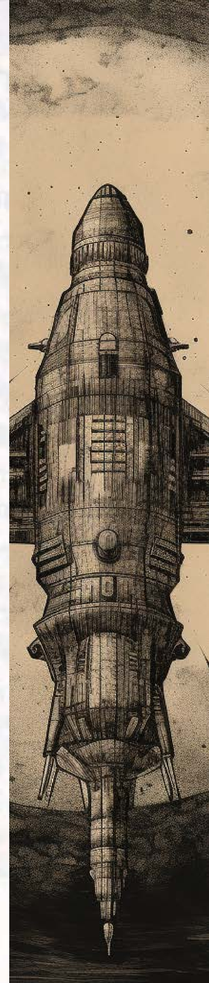
"Registration is expired," the alien says before moving around you and drifting away down the passageway.

Bewildered, you chase the alien down. "I just renewed my registration this morning! There's got to be some mistake!"

The alien stops. "Listen, kid," it says, sparing you another look. "You can't park there unless you have valid registration. I would get that taken care of before we blast it to smithereens with the Orbital Cannon."

Renew your registration for good this time by submitting a `ExtendFootprintTtlOp` via a "classic" Stellar transaction.

In order to claim your NFT pass your `_nft_dest` as a transaction memo.



Skirmish XIV

In order to claim your NFT pass your `_nft_dest` as a transaction memo

Alright. Your registration is finally up to date and it's time to get to the important stuff. These aliens actually seem pretty friendly, and a fellow resembling a lobster even offers you a wave as you traverse the maze of corridors. It's time to admit that you probably won't find Bee just wandering around, so you flag down a passing alien with long green ears to ask for directions.

"Hi there," you say. "I'm looking for Blossom Bernice Breydenblach. Do you know where she might be?"

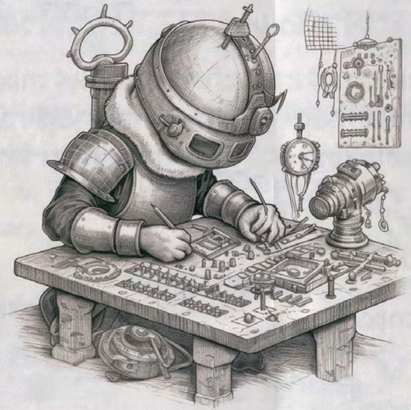
"I am Ambassador Zillwow!" the alien announces.

You blink. "Umm, okay. That's great. But I'm looking for Blossom Bernice Breydenblach."

Ambassador Zillwow turns and points to a window looking out into starry space where you see a strawberry blonde-headed figure drift by.

She waves frantically, and you realize that this is Bee! Horrified, you know you must save her. Submit a Stellar transaction with a singular `RestoreFootprintOp` to save Bee and restore her back to the space station.

Prepare yourself.



RPCiege WILL RETURN.

Need some help in the meantime? Check out the Soroban documentation, join the developer Discord, and follow Soroban on Twitter.

Legal Disclaimer: RPCiege tasks are educational in nature and provide a structured sandbox environment for learning how to use Stellar software. All RPCiege tasks are performed on a testnet using test assets. Nothing in these RPCiege instructions should be construed as financial, legal or investment advice. Separately, if you choose to interact with AMM functionality and liquidity pools using real assets on Stellar mainnet then you should ensure you understand the technology, the assets and that you are aware of the risks involved in such operations. Remember, the value of crypto assets can be extremely volatile and unpredictable, which can result in significant losses in a short time including possibly a loss of total value.

PRINTED IN JAPAN