Debugging Programs

Solana programs run on-chain, so debugging them in the wild can be challenging. To make debugging programs easier, developers can write unit tests that directly test their program's execution via the Solana runtime, or run a local cluster that will allow RPC clients to interact with their program.

Running unit tests#

- Testing with Rust
- Testing with C

Logging#

During program execution both the runtime and the program log status and error messages.

For information about how to log from a program see the language specific documentation:

- Logging from a Rust program
- Logging from a C program

When running a local cluster the logs are written to stdout as long as they are enabled via theRUST_LOG log mask. From the perspective of program development it is helpful to focus on just the runtime and program logs and not the rest of the cluster logs. To focus in on program specific information the following log mask is recommended:

export
RUST_LOG=solana_runtime::system_instruction_processor=trace,solana_runtime::message_processor=info,solana_bpf_loader=debug,solana_rbpf=debug
Log messages coming directly from the program (not the runtime) will be displayed in the form:

Program log:

Error Handling#

The amount of information that can be communicated via a transaction error is limited but there are many points of possible failures. The following are possible failure points and information about what errors to expect and where to get more information:

- The SBF loader may fail to parse the program, this should not happen since the
- loader has alreadyfinalized
- the program's account data.* InstructionError::InvalidAccountData
-
  - will be returned as part of the
-
  - transaction error.
- The SBF loader may fail to setup the program's execution environment* InstructionError::Custom(0x0b9f_0001)
-
  - will be returned as part of the
-
  - transaction error. "0x0b9f_0001" is the hexadecimal representation ofVirtualMachineCreationFailed
-
  - .
- The SBF loader may have detected a fatal error during program executions
- (things like panics, memory violations, system call errors, etc...)* InstructionError::Custom(0x0b9f_0002)
-
  - will be returned as part of the
-
  - transaction error. "0x0b9f_0002" is the hexadecimal representation ofVirtualMachineFailedToRunProgram
-
  - .
- The program itself may return an error* InstructionError::Custom()
-
  - will be returned. The "user
-
  - defined value" must not conflict with any of thebuiltin runtime program errors
-
  - .
-
  - Programs typically use enumeration types to define error codes starting at
-
  - zero so they won't conflict.

In the case ofVirtualMachineFailedToRunProgram errors, more information about the specifics of what failed are written to theprogram's execution logs .

For example, an access violation involving the stack will look something like this:

SBF program 4uQeVj5tqViQh7yWWGStvkEG1Zmhx6uasJtWCJziofM failed: out of bounds memory store (insn #615), addr 0x200001e38/8

Monitoring Compute Budget Consumption#

The program can log the remaining number of compute units it will be allowed before program execution is halted. Programs can use these logs to wrap operations they wish to profile.

- Log the remaining compute units from a Rust program
- Log the remaining compute units from a C program

Seecompute budget for more information.

ELF Dump#

The SBF shared object internals can be dumped to a text file to gain more insight into a program's composition and what it may be doing at runtime.

- Create a dump file of a Rust program
- Create a dump file of a C program

Instruction Tracing#

During execution the runtime SBF interpreter can be configured to log a trace message for each SBF instruction executed. This can be very helpful for things like pin-pointing the runtime context leading up to a memory access violation.

The trace logs together with theELF dump can provide a lot of insight (though the traces produce a lot of information).

To turn on SBF interpreter trace messages in a local cluster configure thesolana_rbpf level inRUST_LOG totrace . For example:

export RUST_LOG=solana_rbpf=trace

Source level debugging#

Source level debugging of on-chain programs written in Rust or C can be done using theprogram run subcommand ofsolana-ledger-tool , and lldb, distributed with Solana Rust and Clang compiler binary package platform-tools.

Thesolana-ledger-tool program run subcommand loads a compiled on-chain program, executes it in RBPF virtual machine and runs a gdb server that accepts incoming connections from LLDB or GDB. Once lldb is connected tosolana-ledger-tool gdbserver, it can control execution of an on-chain program. Runsolana-ledger-tool program run --help for an example of specifying input data for parameters of the program entrypoint function.

To compile a program for debugging use cargo-build-sbf build utility with the command line option--debug . The utility will generate two loadable files, one a usual loadable module with the extension.so , and another the same loadable module but containing Dwarf debug information, a file with extension.debug .

To execute a program in debugger, runsolana-ledger-tool program run with-e debugger command line option. For example, a crate named 'helloworld' is compiled and an executable program is built intarget/deploy directory. There should be three files in that directory

- helloworld-keypair.json -- a keypair for deploying the program,
- helloworld.debug -- a binary file containing debug information,
- helloworld.so -- an executable file loadable into the virtual machine. The
- command line for runningsolana-ledger-tool
- would be something like this

solana-ledger-tool program run -l test-ledger -e debugger target/deploy/helloworld.so Note thatsolana-ledger-tool always loads a ledger database. Most on-chain programs interact with a ledger in some manner. Even if for debugging purpose a ledger is not needed, it has to be provided tosolana-ledger-tool . A minimal ledger database can be created by runningsolana-test-validator , which creates a ledger intest-ledger subdirectory.

In debugger modesolana-ledger-tool program run loads an.so file and starts listening for an incoming connection from a debugger

Waiting for a Debugger connection on "127.0.0.1:9001"... To connect tosolana-ledger-tool and execute the program, run lldb. For debugging rust programs it may be beneficial to run solana-lldb wrapper to lldb, i.e. at a new shell prompt (other than the one used to startsolana-ledger-tool) run the command:

solana-lldb This script is installed in platform-tools path. If that path is not added toPATH environment variable, it may be necessary to specify the full path, e.g.

~/.cache/solana/v1.35/platform-tools/llvm/bin/solana-lldb After starting the debugger, load the .debug file by entering the following command at the debugger prompt

(lldb) file target/deploy/helloworld.debug If the debugger finds the file, it will print something like this

Current executable set to '/path/helloworld.debug' (bpf). Now, connect to the gdb server thatsolana-ledger-tool implements, and debug the program as usual. Enter the following command at lldb prompt

(lldb) gdb-remote 127.0.0.1:9001 If the debugger and the gdb server establish a connection, the execution of the program will be stopped at the entrypoint function, and lldb should print several lines of the source code around the entrypoint function signature. From this point on, normal lldb commands can be used to control execution of the program being debugged.

Debugging in an IDE#

To debug on-chain programs in Visual Studio IDE, install the CodeLLDB extension. Open CodeLLDB Extension Settings. In Advanced settings change the value ofLldb: Library field to the path oflibLldb.so (or libLldb.dylib on macOS). For example on Linux a possible path to Solana customized lldb can be/home//.cache/solana/v1.33/platform-tools/llvm/lib/libLldb.so. where is your Linux system username. This can also be added directly to~/.config/Code/User/settings.json file, e.g.

{ "lldb.library": "/home//.cache/solana/v1.35/platform-tools/llvm/lib/libLldb.so" } In.vscode subdirectory of your on-chain project, create two files

First file istasks.json with the following content

{ "version": "2.0.0", "tasks": [{ "label": "build", "type": "shell", "command": "cargo build-sbf --debug", "problemMatcher": [], "group": { "kind": "build", "isDefault": true } }, { "label": "solana-debugger", "type": "shell", "command": "solana-ledger-tool program run -l test-ledger -e debugger {workspaceFolder}/target/deploy/helloworld.so" }] } The first task is to build the on-chain program using cargo-build-sbf utility. The second task is to runsolana-ledger-tool program run in debugger mode.

Another file islaunch.json with the following content

{ "version": "0.2.0", "configurations": [{ "type": "lldb", "request": "custom", "name": "Debug", "targetCreateCommands": ["target create {workspaceFolder}/target/deploy/helloworld.debug"], "processCreateCommands": ["gdb-remote 127.0.0.1:9001"] }] } This file specifies how to run debugger and to connect it to the gdb server implemented bysolana-ledger-tool .

To start debugging a program, first build it by running the build task. The next step is to runsolana-debugger task. The tasks specified intasks.json file are started fromTerminal >> Run Task... menu of VSCode. Whensolana-ledger-tool is running and listening from incoming connections, it's time to start the

debugger. Launch it from VSCodeRun and Debug menu. If everything is set up correctly, VSCode will start a debugging session and the program execution should stop on the entrance into theentrypoint function.