Debugging Bitcoin
Fabian Jahr
DISCLAIMER

This drama is inspired by true events but it is not a documentary/biography of any character depicted in the drama. No scenes should be construed to represent a true or accurate recreation of the actual events that transpired. The story and the relationships between the characters depicted in the drama have been fictionalized. Any insinuation or resemblance of any character to any person, dead or alive, or the resemblance of the drama story to any real story is purely coincidental.

Cigarette smoking, alcohol and tobacco consumption are injurious to health.
Welcome to Bitcoin
First issues

```
LogPrintf("@@@@@@@@@@@@");
std::cout << "################" << std::endl;
```

```
$ cat ~/Library/Application\ Support/Bitcoin/debug.log | grep @@
```

wat?
Let's ask in the chat

1337hax0r
Hey, the logging doesn't work!!!
Helpful senior dev is helpful

```
$ bitcoin-cli -regtest getdifficulty
4.656542373906925e-10
```

```
std::out
2019-08-05T17:36:14Z Received a POST request for / from 127.0.0.1:65457
2019-08-05T17:36:14Z ThreadRPCServer method=getdifficulty user=satoshi
2019-08-05T17:36:14Z >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
```

```
regtest/debug.log
$ sudo cat ~/Library/Application\ Support/Bitcoin/regtest/debug.log | grep "error"
2019-08-05T17:27:19Z "Adding fixed seed nodes as DNS doesn't seem to be available."
2019-08-05T17:34:47Z "tor: Error connecting to Tor control socket"
2019-08-05T17:36:14Z "tor: Error connecting to Tor control socket"
```
Moving on to unit tests

To print messages in the unit tests, you cannot use `LogPrintf`. Instead use `BOOST_TEST_MESSAGE` and `BOOST_CHECK_MESSAGE` to print messages from the tests. You will have to run the test binary directly (`src/test/test_bitcoin`) with `--log_level=all` to see the messages.

To print from the source files themselves, you could use `fprintf()` and print your messages to stderr.
That was kind of easy
What about using a debugger?

---

gdb or lldb on macOS

Start debugger with an executable

Set breakpoints

Run the executable from the debugger

Inspect variables, step through lines etc.
This is pretty cool!

```bash
$ lldb src/test/test_bitcoin
(lldb) target create "src/test/test_bitcoin"
Current executable set to 'src/test/test_bitcoin' (x86_64).
(lldb) b test/blockchain_tests.cpp:48
Breakpoint 1: 5 locations.
(lldb) run --run-test=blockchain_tests
Test_bitcoin was compiled with optimization - stepping may behave oddly.
Process 46577 stopped
* thread #1, queue = 'com.apple.main-thread', stop reason = breakpoint 1
  frame #0: 0x0000000100093d42 test_bitcoin`GLOBAL__sub_I_blockchain
  45 46  BOOST_FIXTURE_TEST_SUITE(blockchain_tests, BasicTestingSetup)
  47 -> 48  BOOST_AUTO_TEST_CASE(get_difficulty_for_very_low_target)
  49  {
  50   TestDifficulty(0x1f11111, 0.000001);
  51 } 
Target 0: (test_bitcoin) stopped.
```
Should not be too hard for functional tests...

Using Python

Logging
  - `self.log.debug()`

Debugging
  - `import pdb; pdb.set_trace()`

But what about debugging the C++ code?
But where is the executable?

Functional tests launch our `bitcoind` themselves using a temp folder as datadir.

That means we can not simply start it ourselves.

We need a gameplan!
Gameplan

1. Start the functional test directly (not using `test_runner.py`) and let them start the `bitcoind` process.
2. Pause the functional tests with `pdb.set_trace()`.
3. Find the running `bitcoind` process, attach to it using `lldb` and setting breakpoints.
4. Then let the test continue (continue in `pdb`) and let it run into our `lldb` breakpoints.
5. Optional: May want to remove 60s timeout.
Major key to success: Context awareness
# Debugging contexts

<table>
<thead>
<tr>
<th>Manual</th>
<th>Test driver</th>
<th>Bitcoind context</th>
</tr>
</thead>
</table>
| -      | `bitcoin-cli/rpc` | - Path: your own bitcoin path  
- Log: `ENV/debug.log`  
- Debug: run `bitcoind` with `lldb` |

<table>
<thead>
<tr>
<th>Unit tests</th>
<th>Test driver</th>
<th>Bitcoind context</th>
</tr>
</thead>
</table>
| -          | `src/test/test_bitcoin` | - Path: `/var/`  
- Log: to `std::out` with LibBoost  
- Debug: Run `test_bitcoin` with LLDB |

<table>
<thead>
<tr>
<th>Functional tests</th>
<th>Test driver</th>
<th>Bitcoind context</th>
</tr>
</thead>
</table>
| - test/functional/test_runner.py (or the test directly)  
- Log: `self.log.print()`  
- Debug: `pdb` | - Path: `/var/` with `--no-cleanup`  
- Log: temporary `debug.log` with consolidation tool  
- Debug: `pdb + lldb` |
Things left out

Install `ccache`

Compiler flags
- Disable optimizations (`-O0`)
- Remove parts you don’t need, e.g. `--without-gui` for example

Segfault tools
- Core dumps
  - Need to activate with `ulimit -c unlimited` and then run in same terminal session
  - Find them in `/cores/*`
  - Make sure to clean up afterwards
- `valgrind`
  - Inspections, used similar to `lldb`
  => work in progress
  => help me with linux version!

Thank you and questions?