Fuzzing Bitcoin Core using libFuzzer

Quickstart guide

To quickly get started fuzzing Bitcoin Core using libFuzzer:

```
$ git clone https://github.com/bitcoin/bitcoin
$ cd bitcoin/
$ ./autogen.sh
$ CC=clang CXX=clang++ ./configure --enable-fuzz --with-sanitizers=address,fuzzer,undefined
# macOS users: If you have problem with this step then make sure to read "macOS hints for
# libFuzzer" on https://github.com/bitcoin/bitcoin/blob/master/doc/fuzzing.md#macos-hints-f
$ make
$ FUZZ=process_message src/test/fuzz/fuzz
# abort fuzzing using ctrl-c
```

There is also a runner script to execute all fuzz targets. Refer to `./test/fuzz/test_runner.py --help` for more details.

Overview of Bitcoin Core fuzzing

Google has a good overview of fuzzing in general, with contributions from key architects of some of the most-used fuzzers. This paper includes an external overview of the status of Bitcoin Core fuzzing, as of summer 2021. John Regehr provides good advice on writing code that assists fuzzers in finding bugs, which is useful for developers to keep in mind.

Fuzzing harnesses and output

`process_message` is a fuzzing harness for the `ProcessMessage(...)` function (`net_processing`). The available fuzzing harnesses are found in `src/test/fuzz/`.

The fuzzer will output `NEW` every time it has created a test input that covers new areas of the code under test. For more information on how to interpret the fuzzer output, see the libFuzzer documentation.

If you specify a corpus directory then any new coverage increasing inputs will be saved there:

```
$ mkdir -p process_message-seeded-from-thin-air/
$ FUZZ=process_message src/test/fuzz/fuzz process_message-seeded-from-thin-air/
INFO: Seed: 840522292
INFO: Loaded 1 modules   (424174 inline 8-bit counters): 424174 [0x55e121ef9ab8, 0x55e121f61
INFO: Loaded 1 PC tables (424174 PCs): 424174 [0x55e121f613a8,0x55e1225da288),
INFO:        0 files found in process_message-seeded-from-thin-air/
INFO: -max_len is not provided; libFuzzer will not generate inputs larger than 4096 bytes
INFO: A corpus is not provided, starting from an empty corpus
#2      INITED cov: 94 ft: 95 corp: 1/1b exec/s: 0 rss: 150Mb
```

```
#3      NEW    cov: 95 ft: 96 corp: 2/3b lim: 4 exec/s: 0 rss: 150Mb L: 2/2 MS: 1 InsertByt
#4      NEW    cov: 96 ft: 98 corp: 3/7b lim: 4 exec/s: 0 rss: 150Mb L: 4/4 MS: 1 CrossOver
#21     NEW    cov: 96 ft: 100 corp: 4/11b lim: 4 exec/s: 0 rss: 150Mb L: 4/4 MS: 2 ChangeB
#324    NEW    cov: 101 ft: 105 corp: 5/12b lim: 6 exec/s: 0 rss: 150Mb L: 6/6 MS: 5 CrossO
#1239   REDUCE cov: 102 ft: 106 corp: 6/24b lim: 14 exec/s: 0 rss: 150Mb L: 13/13 MS: 5 Cha
#1272   REDUCE cov: 102 ft: 106 corp: 6/23b lim: 14 exec/s: 0 rss: 150Mb L: 12/12 MS: 3 Cha
        NEW_FUNC[1/677]: 0x55e11f456690 in std::_Function_base::~_Function_base() /usr/lib/g
        NEW_FUNC[2/677]: 0x55e11f465800 in CDataStream::CDataStream(std::vector<unsigned cha
#2125   REDUCE cov: 4820 ft: 4867 corp: 7/29b lim: 21 exec/s: 0 rss: 155Mb L: 6/12 MS: 2 Co
        NEW_FUNC[1/9]: 0x55e11f64d790 in std::_Rb_tree<uint256, std::pair<uint256 const, sto
        NEW_FUNC[2/9]: 0x55e11f64d870 in std::_Rb_tree<uint256, std::pair<uint256 const, sto
#2228   NEW    cov: 4898 ft: 4971 corp: 8/35b lim: 21 exec/s: 0 rss: 156Mb L: 6/12 MS: 3 Ero
        NEW_FUNC[1/5]: 0x55e11f46df70 in std::enable_if<__and_<std::allocator_traits<zero_a
        NEW_FUNC[2/5]: 0x55e11f477390 in std::vector<unsigned char, std::allocator<unsigned
#2456   NEW    cov: 4933 ft: 5042 corp: 9/55b lim: 21 exec/s: 0 rss: 160Mb L: 20/20 MS: 3 C
#2467   NEW    cov: 4933 ft: 5043 corp: 10/76b lim: 21 exec/s: 0 rss: 161Mb L: 21/21 MS: 1
#4215   NEW    cov: 4941 ft: 5129 corp: 17/205b lim: 29 exec/s: 4215 rss: 350Mb L: 29/29 MS
#4567   REDUCE cov: 4941 ft: 5129 corp: 17/204b lim: 29 exec/s: 4567 rss: 404Mb L: 24/29 MS
#6642   NEW    cov: 4941 ft: 5138 corp: 18/244b lim: 43 exec/s: 2214 rss: 450Mb L: 43/43 MS
# abort fuzzing using ctrl-c
$ ls process_message-seeded-from-thin-air/
349ac589fc66a09abc0b72bb4ae445a7a19e2cd8 4df479f1f421f2ea64b383cd4919a272604087a7
a640312c98dcc55d6744730c33e41c5168c55f09 b135de16e4709558c0797c15f86046d31c5d86d7
c000f7b41b05139de8b63f4cbf7d1ad4c6e2aa7f fc52cc00ec1eb1c08470e69f809ae4993fa70082
$ cat --show-nonprinting process_message-seeded-from-thin-air/349ac589fc66a09abc0b72bb4ae445
block^@M-^?M-^?M-^?M-^?M-^?M-^?nM-^?M-^?
```

In this case the fuzzer managed to create a `block` message which when passed to `ProcessMessage(...)` increased coverage.

It is possible to specify `bitcoind` arguments to the `fuzz` executable. Depending on the test, they may be ignored or consumed and alter the behavior of the test. Just make sure to use double-dash to distinguish them from the fuzzer's own arguments:

```
$ FUZZ=address_deserialize_v2 src/test/fuzz/fuzz -runs=1 fuzz_seed_corpus/address_deserializ
```

Fuzzing corpora

The project's collection of seed corpora is found in the `bitcoin-core/qa-assets` repo.

To fuzz `process_message` using the `bitcoin-core/qa-assets` seed corpus:

```
$ git clone https://github.com/bitcoin-core/qa-assets
$ FUZZ=process_message src/test/fuzz/fuzz qa-assets/fuzz_seed_corpus/process_message/
INFO: Seed: 1346407872
INFO: Loaded 1 modules   (424174 inline 8-bit counters): 424174 [0x55d8a9004ab8, 0x55d8a906
```

```
INFO: Loaded 1 PC tables (424174 PCs): 424174 [0x55d8a906c3a8,0x55d8a96e5288),
INFO:      991 files found in qa-assets/fuzz_seed_corpus/process_message/
INFO: -max_len is not provided; libFuzzer will not generate inputs larger than 4096 bytes
INFO: seed corpus: files: 991 min: 1b max: 1858b total: 288291b rss: 150Mb
#993    INITED cov: 7063 ft: 8236 corp: 25/3821b exec/s: 0 rss: 181Mb
...
```

Run without sanitizers for increased throughput

Fuzzing on a harness compiled with `--with-sanitizers=address,fuzzer,undefined` is good for finding bugs. However, the very slow execution even under libFuzzer will limit the ability to find new coverage. A good approach is to perform occasional long runs without the additional bug-detectors (configure `--with-sanitizers=fuzzer`) and then merge new inputs into a corpus as described in the qa-assets repo (https://github.com/bitcoin-core/qa-assets/blob/main/.github/PULL_REQUEST_TEMPLATE.md). Patience is useful; even with improved throughput, libFuzzer may need days and 10s of millions of executions to reach deep/hard targets.

Reproduce a fuzzer crash reported by the CI

- `cd` into the `qa-assets` directory and update it with `git pull qa-assets`
- locate the crash case described in the CI output, e.g. `Test unit written to ./crash-1bc91feec9fc00b107d97dc225a9f2cdaa078eb6`
- make sure to compile with all sanitizers, if they are needed (fuzzing runs more slowly with sanitizers enabled, but a crash should be reproducible very quickly from a crash case)
- run the fuzzer with the case number appended to the seed corpus path: `FUZZ=process_message src/test/fuzz/fuzz qa-assets/fuzz_seed_corpus/process_message/1b`

Submit improved coverage

If you find coverage increasing inputs when fuzzing you are highly encouraged to submit them for inclusion in the `bitcoin-core/qa-assets` repo.

Every single pull request submitted against the Bitcoin Core repo is automatically tested against all inputs in the `bitcoin-core/qa-assets` repo. Contributing new coverage increasing inputs is an easy way to help make Bitcoin Core more robust.

macOS hints for libFuzzer

The default Clang/LLVM version supplied by Apple on macOS does not include fuzzing libraries, so macOS users will need to install a full version, for example using `brew install llvm`.

Should you run into problems with the address sanitizer, it is possible you may need to run `./configure` with `--disable-asm` to avoid errors with certain assembly code from Bitcoin Core's code. See developer notes on sanitizers for more information.

You may also need to take care of giving the correct path for `clang` and `clang++`, like `CC=/path/to/clang CXX=/path/to/clang++` if the non-systems `clang` does not come first in your path.

Full configure that was tested on macOS Catalina with `brew` installed `llvm`:

```
./configure --enable-fuzz --with-sanitizers=fuzzer,address,undefined CC=/usr/local/opt/llvm/
```

Read the libFuzzer documentation for more information. This libFuzzer tutorial might also be of interest.

Fuzzing Bitcoin Core using afl++

Quickstart guide

To quickly get started fuzzing Bitcoin Core using afl++:

```
$ git clone https://github.com/bitcoin/bitcoin
$ cd bitcoin/
$ git clone https://github.com/AFLplusplus/AFLplusplus
$ make -C AFLplusplus/ source-only
$ ./autogen.sh
# If afl-clang-lto is not available, see
# https://github.com/AFLplusplus/AFLplusplus#a-selecting-the-best-afl-compiler-for-instrumen
$ CC=$(pwd)/AFLplusplus/afl-clang-lto CXX=$(pwd)/AFLplusplus/afl-clang-lto++ ./configure --e
$ make
# For macOS you may need to ignore x86 compilation checks when running "make". If so,
# try compiling using: AFL_NO_X86=1 make
$ mkdir -p inputs/ outputs/
$ echo A > inputs/thin-air-input
$ FUZZ=bech32 AFLplusplus/afl-fuzz -i inputs/ -o outputs/ -- src/test/fuzz/fuzz
# You may have to change a few kernel parameters to test optimally - afl-fuzz
# will print an error and suggestion if so.
```

Read the afl++ documentation for more information.

Fuzzing Bitcoin Core using Honggfuzz

Quickstart guide

To quickly get started fuzzing Bitcoin Core using Honggfuzz:

```
$ git clone https://github.com/bitcoin/bitcoin
$ cd bitcoin/
```

```
$ ./autogen.sh
$ git clone https://github.com/google/honggfuzz
$ cd honggfuzz/
$ make
$ cd ..
$ CC=$(pwd)/honggfuzz/hfuzz_cc/hfuzz-clang CXX=$(pwd)/honggfuzz/hfuzz_cc/hfuzz-clang++ ./co
$ make
$ mkdir -p inputs/
$ FUZZ=process_message honggfuzz/honggfuzz -i inputs/ -- src/test/fuzz/fuzz
```

Read the Honggfuzz documentation for more information.

Fuzzing the Bitcoin Core P2P layer using Honggfuzz NetDriver

Honggfuzz NetDriver allows for very easy fuzzing of TCP servers such as Bitcoin Core without having to write any custom fuzzing harness. The `bitcoind` server process is largely fuzzed without modification.

This makes the fuzzing highly realistic: a bug reachable by the fuzzer is likely also remotely triggerable by an untrusted peer.

To quickly get started fuzzing the P2P layer using Honggfuzz NetDriver:

```
$ mkdir bitcoin-honggfuzz-p2p/
$ cd bitcoin-honggfuzz-p2p/
$ git clone https://github.com/bitcoin/bitcoin
$ cd bitcoin/
$ ./autogen.sh
$ git clone https://github.com/google/honggfuzz
$ cd honggfuzz/
$ make
$ cd ..
$ CC=$(pwd)/honggfuzz/hfuzz_cc/hfuzz-clang \
      CXX=$(pwd)/honggfuzz/hfuzz_cc/hfuzz-clang++ \
      ./configure --disable-wallet --with-gui=no \
                  --with-sanitizers=address,undefined
$ git apply << "EOF"
diff --git a/src/bitcoind.cpp b/src/bitcoind.cpp
index 455a82e39..2faa3f80f 100644
--- a/src/bitcoind.cpp
+++ b/src/bitcoind.cpp
@@ -158,7 +158,11 @@ static bool AppInit(int argc, char* argv[])
     return fRet;
 }

+#ifdef HFND_FUZZING_ENTRY_FUNCTION_CXX
+HFND_FUZZING_ENTRY_FUNCTION_CXX(int argc, char* argv[])
```

```
+#else
 int main(int argc, char* argv[])
+#endif
 {
 #ifdef WIN32
     util::WinCmdLineArgs winArgs;
diff --git a/src/net.cpp b/src/net.cpp
index cf987b699..636a4176a 100644
--- a/src/net.cpp
+++ b/src/net.cpp
@@ -709,7 +709,7 @@ int V1TransportDeserializer::readHeader(const char *pch, unsigned int nB
     }

     // Check start string, network magic
-    if (memcmp(hdr.pchMessageStart, m_chain_params.MessageStart(), CMessageHeader::MESSAGE_
+    if (false && memcmp(hdr.pchMessageStart, m_chain_params.MessageStart(), CMessageHeader:
         LogPrint(BCLog::NET, "HEADER ERROR - MESSAGESTART (%s, %u bytes), received %s, peer
         return -1;
     }
@@ -768,7 +768,7 @@ Optional<CNetMessage> V1TransportDeserializer::GetMessage(const std::chr
     RandAddEvent(ReadLE32(hash.begin()));

     // Check checksum and header command string
-    if (memcmp(hash.begin(), hdr.pchChecksum, CMessageHeader::CHECKSUM_SIZE) != 0) {
+    if (false && memcmp(hash.begin(), hdr.pchChecksum, CMessageHeader::CHECKSUM_SIZE) != 0)
         LogPrint(BCLog::NET, "CHECKSUM ERROR (%s, %u bytes), expected %s was %s, peer=%d\n"
                  SanitizeString(msg->m_command), msg->m_message_size,
                  HexStr(Span<uint8_t>(hash.begin(), hash.begin() + CMessageHeader::CHECKSUM
EOF
$ make -C src/ bitcoind
$ mkdir -p inputs/
$ honggfuzz/honggfuzz --exit_upon_crash --quiet --timeout 4 -n 1 -Q \
      -E HFND_TCP_PORT=18444 -f inputs/ -- \
          src/bitcoind -regtest -discover=0 -dns=0 -dnsseed=0 -listenonion=0 \
                       -nodebuglogfile -bind=127.0.0.1:18444 -logthreadnames \
                       -debug
```

Fuzzing Bitcoin Core using Eclipser (v1.x)

Quickstart guide

To quickly get started fuzzing Bitcoin Core using Eclipser v1.x:

```
$ git clone https://github.com/bitcoin/bitcoin
$ cd bitcoin/
$ sudo vim /etc/apt/sources.list # Uncomment the lines starting with 'deb-src'.
```

```
$ sudo apt-get update
$ sudo apt-get build-dep qemu
$ sudo apt-get install libtool libtool-bin wget automake autoconf bison gdb
```

At this point, you must install the .NET core. The process differs, depending on your Linux distribution. See this link for details. On ubuntu 20.04, the following should work:

```
$ wget -q https://packages.microsoft.com/config/ubuntu/20.04/packages-microsoft-prod.deb
$ sudo dpkg -i packages-microsoft-prod.deb
$ rm packages-microsoft-prod.deb
$ sudo apt-get update
$ sudo apt-get install -y dotnet-sdk-2.1
```

You will also want to make sure Python is installed as `python` for the Eclipser install to succeed.

```
$ git clone https://github.com/SoftSec-KAIST/Eclipser.git
$ cd Eclipser
$ git checkout v1.x
$ make
$ cd ..
$ ./autogen.sh
$ ./configure --enable-fuzz
$ make
$ mkdir -p outputs/
$ FUZZ=bech32 dotnet Eclipser/build/Eclipser.dll fuzz -p src/test/fuzz/fuzz -t 36000 -o outp
```

This will perform 10 hours of fuzzing.

To make further use of the inputs generated by Eclipser, you must first decode them:

```
$ dotnet Eclipser/build/Eclipser.dll decode -i outputs/testcase -o decoded_outputs
```

This will place raw inputs in the directory `decoded_outputs/decoded_stdins`. Crashes are in the `outputs/crashes` directory, and must be decoded in the same way.

Fuzzing with Eclipser will likely be much more effective if using an existing corpus:

```
$ git clone https://github.com/bitcoin-core/qa-assets
$ FUZZ=bech32 dotnet Eclipser/build/Eclipser.dll fuzz -p src/test/fuzz/fuzz -t 36000 -i qa-a
```

Note that fuzzing with Eclipser on certain targets (those that create 'full nodes', e.g. `process_message*`) will, for now, slowly fill `/tmp/` with improperly cleaned-up files, which will cause spurious crashes. See this proposed patch for more information.

Read the Eclipser documentation for v1.x for more details on using Eclipser.

OSS-Fuzz

Bitcoin Core participates in Google's OSS-Fuzz program, which includes a dashboard of publicly disclosed vulnerabilities. Generally, we try to disclose vulnerabilities as soon as possible after they are fixed to give users the knowledge they need to be protected. However, because Bitcoin is a live P2P network, and not just standalone local software, we might not fully disclose every issue within Google's standard 90-day disclosure window if a partial or delayed disclosure is important to protect users or the function of the network.

OSS-Fuzz also produces a fuzzing coverage report.