Monero Fuzzing Audit

Monero Fuzzing Report

David Korczynski, Adam korczynski, Arthur Chan

17-08-2025

About Ada Logics

Contents

1 Executive Summary

This report contains the results from a fuzzing audit of Monero. The main goal of the fuzzing audit was to enable fuzzing of Monero's RPC endpoints and do so by way of OSS-Fuzz. The target was to enable fuzzing of 75% of Monero's RPC handlers in terms of code coverage.

The main achievements from this audit is twofold:

1) A libFuzzer-based harness that targets Monero's RPC handlers and that is integrated into Monero's OSS-Fuzz project. At the time of writing, this harness exists in a pull request submitted to the Monero repository.
2) A python-based fuzzing harness that targets the `monerod` server directly by way of http requests on a local machine. This is an end-to-end harness that targets the `monerod` binary directly, and in our set up we have enabled ASAN-based compilation of `monerod` as well as coverage collection.

The primary libfuzzer harness found two issues during the audit, and has been run consecutively for 4 days on our machines. Following the 4 days of consecutive run, the fuzzer still routinely explores new coverage, and, consequently, our prediction is that once the PR is merged and the fuzzer starts running on OSS-Fuzz it is likely that more issues will be found with the compute resources offered by OSS-Fuzz.

This works was supported by the MAGIC Monero Fund grants, and we would like to thank the entire team for their collaboration during this engagement.

2 Introduction

In this report we go through the fuzzing audit conducted focused around fuzzing Monero's RPC logic. The goal of this audit was to add fuzzing harnesses such that the RPC handlers of Monero were covered by the harness, and that the harness could be run as part of Monero's OSS-Fuzz project. The goals of this audit were to:

1) Write fuzzing harnesses that would cover 75% of Monero's RPC handler entrypoints.
2) Submit the code upstream ot Monero.
3) Produce a written report going over the details of the engagement.

We have created two primary fuzzing approaches for Monero's RPC handlers. First, an OSS-Fuzz based approach which has been submitted to Monero's upstream repository here and an end-to-end based Python approach which is documented in this report.

Both of these approaches are capable of hitting the entrypoints of all the RPC handlers, and both have a respective 87+% function coverage of the `core_rpc_server.cpp` source file, as is shown in this report. We expect once the RPC fuzzer PR is merged and the harness starts running on OSS-Fuzz further increases will happen.

Overall our conclusion is that Monero is a robust codebase. The harnesses found no issues other than the ones reported per email (2 true positives, erroneous). However, we expect the compute of OSS-Fuzz to likely have impact on this, and we generally expect more issues to show up once the RPC harness starts running on OSS-Fuzz.

In this report we will go through the fuzzing approaches we developed.

3 Fuzzing monero RPC logic

This fuzzing targets the `src/rpc` module within Monero's codebase. The objective is to evaluate the robustness and security of the RPC request-handling logic, including all functions reachable from `src/rpc` through direct or indirect invocation. The fuzzer systematically exercises the request/response interfaces and underlying core logic, aiming to uncover vulnerabilities such as memory corruption, logic flaws, unhandled exceptions, and inconsistencies in input validation. This approach ensures comprehensive coverage of the Monero daemon's external RPC interface and its integration with critical code paths.

3.1 Understanding the src/rpc module

The `src/rpc` directory of the Monero codebase encapsulates the daemon's remote procedure call (RPC) infrastructure, which facilitates external interaction with the node for querying blockchain state, managing transactions, and controlling daemon behavior. To better understand and fuzz this subsystem, the RPC components have been analyzed and logically divided into six groups based on their functional roles:

Group 1: HTTP-based RPC This group includes the core_rpc_server and daemon_handler, which implement Monero's support for REST-like and JSON-RPC APIs over HTTP. It handles endpoint definitions, request parsing, parameter validation, error reporting, and JSON response generation.

Files include:

- rpc/core_rpc_server.cpp
- rpc/core_rpc_server.h
- rpc/core_rpc_server_commands_defs.h
- rpc/core_rpc_server_error_codes.h
- rpc/daemon_handler.cpp
- rpc/daemon_handler.h
- rpc/daemon_messages.cpp
- rpc/daemon_messages.h
- rpc/daemon_rpc_version.h

Group 2: ZeroMQ RPC

Used for real-time event publishing to subscribers. Includes the ZMQ publisher and server modules.

Files include:

- rpc/zmq_pub.cpp
- rpc/zmq_pub.h
- rpc/zmq_server.cpp
- rpc/zmq_server.h

Group 3: Transport Layer and Serialization

Handles serialization and shared message structures used across HTTP and ZMQ transports.

Files include:

- rpc/message.cpp
- rpc/message.h
- rpc/message_data_structs.h
- rpc/instanciations.cpp

- rpc/fwd.h

Group 4: RPC Payment API

Implements Monero's RPC payment system, where clients pay to access gated endpoints.

Files include:

- rpc/rpc_payment.cpp
- rpc/rpc_payment.h
- rpc/rpc_payment_costs.h
- rpc/rpc_payment_signature.cpp
- rpc/rpc_payment_signature.h

Group 5: Bootstrap Daemon Forwarding

Allows forwarding of requests to synchronized remote daemons. Uses heuristics for node selection. Files include:

- rpc/bootstrap_daemon.cpp
- rpc/bootstrap_daemon.h
- rpc/bootstrap_node_selector.cpp
- rpc/bootstrap_node_selector.h

3.2 Fuzzing Limitations and Design Trade-offs

Fuzzing the Monero `src`/`rpc` module presents several architectural and environmental constraints due to both the internal design of the Monero daemon and the sandboxed nature of OSS-Fuzz.

Many core_rpc_server and daemon_handler endpoints rely on Monero's runtime state — including chain height, mempool contents, and hardfork versioning. Without simulating a usable chain state, most handlers will exit prematurely. The solution is to launch the daemon in `--regtest` or `--FAKECHAIN` mode and inject a genesis block, enabling most logic to proceed without peer syncing.

Each RPC endpoint defines its own request/response structure. The fuzzing harness must map fuzzed input into valid types for each endpoint — a generic handler is not feasible.

One of the most critical limitations arises from the OSS-Fuzz environment itself. OSS-Fuzz imposes strict controls on external communication to preserve determinism, isolation, and I/O-free operation. This includes blocking loopback traffic — even requests sent to localhost. As a result, the harness must avoid any reliance on runtime HTTP communication, socket forwarding, or external daemon calls. This excludes the use of the actual Monero HTTP server or RPC dispatcher during fuzzing, even if they are run locally within the same container.

Besides the strict control on calls from fuzzing harness to Monero daemon instance, there are two additional consequences from this constraint:

1. Bootstrap Daemon Forwarding Is Inaccessible

Monero supports delegating RPC calls to a configured "bootstrap daemon", another Monero node that is more synced. This logic is triggered internally in many RPC functions by calling use_bootstrap_daemon_if_necessary. The forwarding occurs via internal invoke_http_json_rpc and similar functions, which construct outbound HTTP requests. In fuzzing mode, however, no such forwarding is possible. External HTTP calls are restricted by OSS-Fuzz, even to loopback addresses. Consequently, any RPC handler path that involves fallback to a bootstrap daemon is statically reachable but cannot be meaningfully fuzzed. This affects handlers such as: on_get_info on_get_block_count on_get_connections which attempt to invoke use_bootstrap_daemon_if_necessary early in their execution and exit if forwarding is blocked.

2. P2P Peer Logic Is Inactive

The core_rpc_server relies on a configured P2P layer for various peer-related RPC endpoints. Since no real P2P connections or hosts can be established during fuzzing, a dummy P2P object is injected during harness initialization. This object contains no known peers, resulting in early termination of logic that attempts to enumerate or interact with the peer list.

For example, in:

- `on_get_peer_list`, which depends on peer enumeration loops.
- `on_set_bans`, which calls block_subnet or unblock_subnet

These RPC endpoint functions exit immediately via CRITICAL_REGION_LOCAL because no hosts exist.

The fuzzing harness cannot simulate these peer structures due to external networking restrictions, which leaves these paths unreachable under OSS-Fuzz constraints.

To overcome these architectural and environmental limitations, three distinct approaches were considered and evaluated below.

3.3 Fuzzing approaches

3.3.1 Approach 1: Full monerod Daemon Execution with External RPC Requests

This approach launches monerod as a background process and sends fuzz-generated HTTP requests to its RPC interface.

Pros:

- Enables full-stack testing: HTTP transport, dispatching, core logic.
- Naturally covers network behaviors and bootstrap forwarding.

Cons:

- High startup overhead and persistent filesystem requirements.
- Incompatible with OSS-Fuzz due to restricted I/O and loopback networking.
- Fragile under sandboxing and parallel fuzzing conditions.

3.3.2 Approach 2: Mock HTTP Server and In-Process JSON Fuzzing

An in-process mock server simulates HTTP calls. Fuzzed JSON is injected as fake requests routed to endpoint logic.

Pros:

- Tests serialization and part of the HTTP stack.
- Avoids daemon process, syncing, and disk I/O.

Cons:

- Requires reimplementing routing and validation logic.
- Still dependent on core_rpc_server and dummy blockchain.
- May diverge from actual HTTP behavior.

3.3.3 Approach 3: Direct Function Invocation with Patched Bootstrap Logic

The harness directly instantiates core_rpc_server with dummy dependencies and calls endpoint handlers (e.g. on_get_info, on_send_raw_tx) with fuzzed data. Blockchain state is simulated in memory using FAKECHAIN. All bootstrap logic is patched to prevent external calls.

Pros:

- Fully deterministic and OSS-Fuzz compliant.
- Deep fuzzing of endpoint logic without transport dependencies.
- Enables structured per-endpoint testing.

Cons:

- No coverage of HTTP transport or parsing.
- Requires endpoint-specific fuzzing logic and mapping.
- Needs simulated blockchain and dummy components.

In the following sections, we will describe solutions to approach (3) and (1) from the above categorisations.

4 OSS-Fuzz fuzzer design and architecture

The focus of our engagement was to enable extended Monero fuzzing on OSS-Fuzz. As such, the main contribution is developing a fuzzer harness that targets the RPC code and is able to work within the OSS-Fuzz environment, which is a harness that follows approach 3 in the previous section. This, this harness is focused on direct RTP handler function invocation and it skips transport layers, meeting OSS-Fuzz constraints on networking and I/O.

The harness described here is available in the Monero pull request tests: Add new fuzzer for RPC endpoints. The entrypoint of the harness is in `fuzz_rpc.cpp` which contains the `LLVMFuzzerTestOneInput` fuzzer entrypoint. The overall idea is to use the fuzz data to select a sequence of Monero RPC handlers to call and use the fuzzer data to seed input to each of these handlers. More specifically, the fuzz input determines which RPC endpoints function to test, and then initialises a core_rpc_server with dummy components, simulates a blockchain with dummy blocks and transactions (via regtest option), and then the RPC handlers are invoked.

To enable fuzzing under OSS-Fuzz the following steps were taken:

- A genesis block is injected programmatically during FAKECHAIN init.
- Bootstrap forwarding logic in bootstrap_daemon.h is patched/stubbed.
- Dummy implementations of i_cryptonote_protocol are used.
- Each RPC endpoint must be individually mapped with custom request objects.

The fuzzer has supportive initialization logic in `initialisation.cpp` and `initialisation.h` which provides initialization of a fuzzer-friendly Monero environment, such as a set of blocks and transactions.

The files `rpc_endpoints.cpp` and `rpc_endpoints.h` contain functions that take in a `FuzzedDataProvider` seeded with fuzz data and call the relevant Monero RPC handlers with arguments seeded by fuzz data.

The fuzzer comes in two variants, one with `SAFE` defined and one with `SAFE` undefined. The difference is that the `SAFE` version only fuzzes at subset of the handlers and do not include any initialization for payment-relevant logic, whereas the harness with no `SAFE` macro defined includes all RPC handlers and also initialization for payment relevant logic. These two versions were constructed because we observed various limitations, e.g. timeouts, in some RPC handlers.

4.1 Preparation and Evolution of the Fuzzing Harness

The current fuzzer design was reached through multiple iterative improvements. The development process evolved in response to early limitations and observations about what logic paths could be meaningfully exercised.

4.1.1 Stage 1: Minimal Initialization and Static Endpoint Invocation

The first version of the harness was developed by instantiating the core_rpc_server with a set of default dummy objects, without any chain simulation or real configuration. The goal was to call each RPC endpoint function in isolation using blank request objects and observe behavior.

This basic setup quickly exposed limitations: most RPC functions exited early. Common exit conditions included failure to parse blank request data, missing fields, uninitialized daemon state, and core readiness checks. Without any valid blockchain context, the handlers refused to engage with deeper logic paths, severely limiting coverage. While this version served as a proof of concept, it highlighted the need for a more realistic initialization.

4.1.2 Stage 2: Attempting Synchronization with Live Peers

In an effort to provide the core object with valid chain state, the second iteration attempted to allow syncing from peers. The idea was to use a full node mode and let the daemon fetch blocks from networked nodes, thereby populating the chain with valid data.

This approach failed in the OSS-Fuzz environment. Synchronization was prohibitively slow and often blocked indefinitely due to lack of peer connectivity, timeouts, or sandbox restrictions. Moreover, even if connectivity was possible, the initialization time would violate OSS-Fuzz performance requirements and fail to provide deterministic, reproducible inputs. As such, this idea was discarded in favor of synthetic chain simulation.

4.1.3 Stage 3: Using Regtest Mode and Structured Request Injection

The third phase focused on leveraging Monero's regtest and FAKECHAIN options to boot the core_rpc_server in a controlled test mode. This enabled the core to initialize without external networking or syncing.

At the same time, the harness began injecting random data into endpoint-specific request structures. Rather than calling endpoints with blank input, the fuzzer now generated data tailored to each request type. This approach allowed basic getter functions to operate normally, and coverage improved for non-critical endpoints like get_info or get_height.

However, deeper RPC handlers still exited early. Many required a valid blockchain state, such as multiple blocks, usable transactions, or a working mempool. Simply generating random request fields was insufficient to engage validation logic or reach more meaningful code paths.

4.1.4 Stage 4: Simulating Blockchain State via Random Blocks and Transactions

To overcome this limitation, the harness was upgraded to generate random blocks, miner transactions, and mempool entries. A helper module was introduced to construct valid, if artificial, blocks using Monero's own internal APIs.

This allowed the harness to initialize a fake chain and simulate realistic daemon behavior. As a result, RPC endpoints that depend on chain state — such as on_getblocktemplate, on_get_transactions, or on_send_raw_tx — began to process and return meaningful responses.

This was a major breakthrough in coverage. However, new problems emerged: some endpoints would hang due to slow block validation, poorly formed requests, or internal loops that consumed excessive time. As a result, timeouts became a dominant cause of fuzzing crashes.

4.1.5 Stage 5: Adding Timeout Recovery with SIGALRM Rewind

To avoid losing progress due to timeout-induced hangs, a timeout handling mechanism was added using SIGALRM. Each fuzzing iteration was wrapped in a timeout guard. If the process exceeded a configured threshold, the current execution was aborted and fuzzing continued. This allowed the harness to survive dangerous or slow requests without halting the entire fuzzer. Memory leaks and dangling state from aborted iterations were tolerated since the main focus was on crash discovery and coverage exploration. With this mechanism, more aggressive RPC handlers could be fuzzed safely.

4.1.6 Stage 6: Distinguishing Risky and Safe Endpoint Categories

Through empirical observation, it became clear that some endpoint functions were more prone to crashing, entering long loops, or producing highly variable behavior. To manage this risk, the codebase was updated to define two sets of functions:

Safe functions: generally stable, quick to return, and less dependent on complex state. Risky functions: functions that commonly failed, hung, or required deep chain state.

Two fuzzers were compiled: one with the SAFE macro defined, which included only safe and critical functions, and another that included the full set including risky endpoints. This modular setup allowed stability during long fuzzing runs, while still enabling full logic coverage through dedicated builds.

4.1.7 Stage 7: Introducing Priority Targets and Biased Selection

Despite improved stability, coverage remained uneven. Some endpoints were hit frequently while others were rarely selected. To correct this, the harness introduced a bias selector and prioritized-endpoint category. Each iteration begins by exercising all critical endpoint functions at least once from a third priority function list (e.g., on_submitblock, on_generateblocks, on_get_transactions). After that, random selection is tuned toward less frequently chosen targets, promoting uniform exploration.

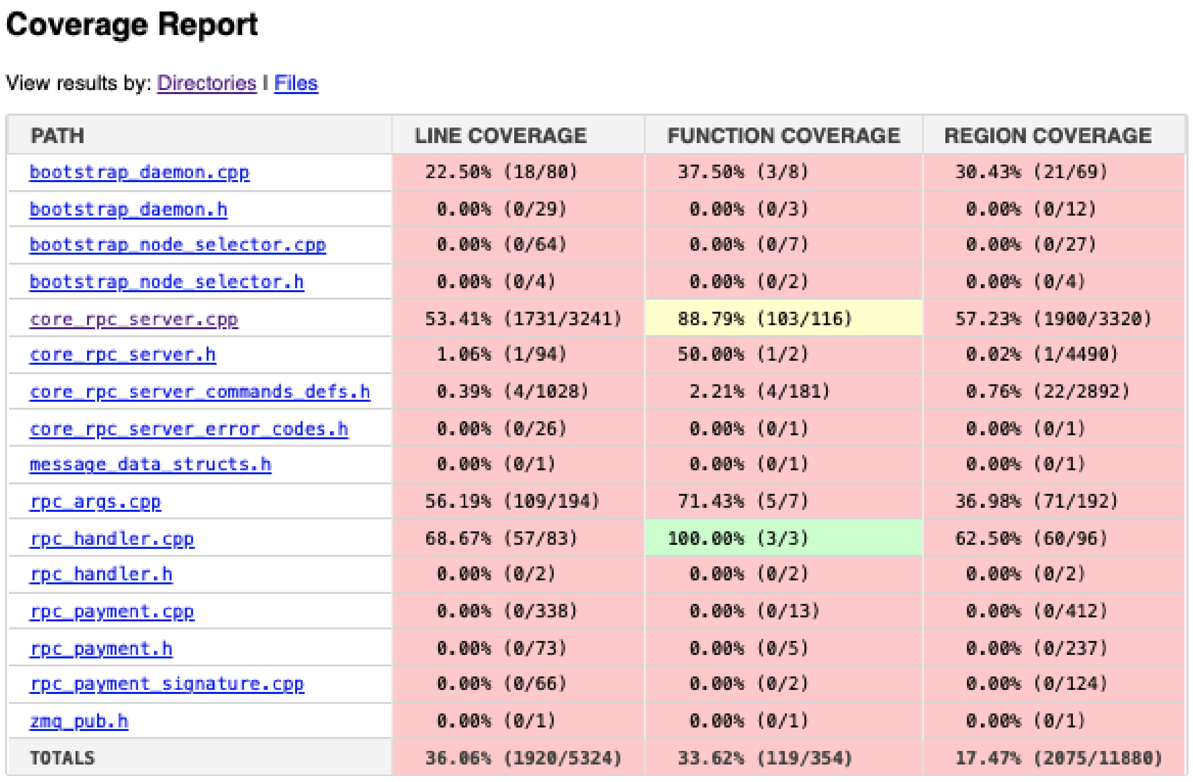
This mechanism increased the breadth of coverage and ensured that all endpoint types received consistent attention over time. The system now reliably covers nearly all core RPC functions given enough iterations.

4.1.8 Stage 8: Expanding to ZMQ Endpoint Fuzzing

Finally, a new fuzzer was introduced to handle Monero's ZMQ endpoints. The fuzz_zmq.cpp harness was added alongside its endpoint wrapper definitions, reusing much of the initialization logic but targeting the pub/sub model instead of standard request/response. This evolutionary path underscores the practical constraints of RPC fuzzing and the need for layered design. By gradually increasing the complex blockchain simulation, refining endpoint handling logic, and controlling execution flow through timeouts and macros, the final harness reaches a robust, extensible state.

4.2 OSS-Fuzz fuzzer harness performance

We have run this fuzzing harness daily during the engagement and once it was ready to be run and tested. To this end, it has run for a long time on our local machine, through various versions of the code. In the current state of the fuzzing harness, we ran it for 96 hours straight and achieved the following coverage, from inside `src/rpc`:



Coverage Report

View results by: Directories | Files

PATH	LINE COVERAGE	FUNCTION COVERAGE	REGION COVERAGE
bootstrap_daemon.cpp	22.50% (18/80)	37.50% (3/8)	30.43% (21/69)
bootstrap_daemon.h	0.00% (0/29)	0.00% (0/3)	0.00% (0/12)
bootstrap_node_selector.cpp	0.00% (0/64)	0.00% (0/7)	0.00% (0/27)
bootstrap_node_selector.h	0.00% (0/4)	0.00% (0/2)	0.00% (0/4)
core_rpc_server.cpp	53.41% (1731/3241)	88.79% (103/116)	57.23% (1900/3320)
core_rpc_server.h	1.06% (1/94)	50.00% (1/2)	0.02% (1/4490)
core_rpc_server_commands_defs.h	0.39% (4/1028)	2.21% (4/181)	0.76% (22/2892)
core_rpc_server_error_codes.h	0.00% (0/26)	0.00% (0/1)	0.00% (0/1)
message_data_structs.h	0.00% (0/1)	0.00% (0/1)	0.00% (0/1)
rpc_args.cpp	56.19% (109/194)	71.43% (5/7)	36.98% (71/192)
rpc_handler.cpp	68.67% (57/83)	100.00% (3/3)	62.50% (60/96)
rpc_handler.h	0.00% (0/2)	0.00% (0/2)	0.00% (0/2)
rpc_payment.cpp	0.00% (0/338)	0.00% (0/13)	0.00% (0/412)
rpc_payment.h	0.00% (0/73)	0.00% (0/5)	0.00% (0/237)
rpc_payment_signature.cpp	0.00% (0/66)	0.00% (0/2)	0.00% (0/124)
zmq_pub.h	0.00% (0/1)	0.00% (0/1)	0.00% (0/1)
TOTALS	36.06% (1920/5324)	33.62% (119/354)	17.47% (2075/11880)

A total of 103 of 116 functions have coverage, and a total of 1731 of 3241 lines are covered. During the 96 hours of fuzzing, we observed continuous edge coverage increase, meaning the fuzzing harness has not plateaued during this run. The harness has a relatively modest execution speed, of around 3-10 executions per second, which means it will need a lot of compute to properly explore. We expect the OSS-Fuzz compute power to have significant impact on the ability of the fuzzer to explore the code, and we expect that both coverage as well as issue count will increase once tests: Add new fuzzer for RPC endpoints lands.

5 End-to-end Python-based fuzzing

In addition to the libfuzzer-based harness that runs on OSS-Fuzz, we developed a pure python-based framework for fuzzing monerod directly. The idea here was the the libfuzzer-based harness relies on some mocking of the environment with respect to how a normal monerod binary would run, and it would be interest to fuzz monerod in it's "normal" state as well.

The main idea behind this harness is to enable fuzzing of monerod with e.g. a fully synchronized blockchain, any options passed to monerod as desired and so on.

We added a set up that enables doing this with Address Sanitizer, as well as have support for coverage collection.

5.1 Architecture of end-to-end fuzzer

The architecture of the end-to-end fuzzer is fairly straightforward since we rely on the monerod binary as is. To support building with sanitizers, we rely on the OSS-Fuzz build script to construct the monerod binary, and the core of the additions to build.sh are:

```
mkdir -p $SRC/monero/monero/build2
cd $SRC/monero/monero/build2

# Add coverage to the build
export CXXFLAGS="$CXXFLAGS --coverage -fprofile-instr-generate -fcoverage-mapping"

cmake -D OSSFUZZ=ON \
      -D STATIC=ON \
      -D BUILD_TESTS=ON \
      -D USE_LTO=OFF \
      -D SANITIZE=ON \
      -D ARCH="default" \
      -DCMAKE_CXX_FLAGS="$CXXFLAGS" \
      -DCMAKE_EXE_LINKER_FLAGS="--coverage" ..
make -j$(nproc) -C src/daemon
cp bin/monerod $OUT/monerod
mkdir -p $SRC/monero/monero/tools
cp -r $SRC/monero_rpc_serialiser $SRC/monero/monero/tools
```

To this end, we build the monerod with both ASAN instrumentation and covergae instrumentation make the workflow simpler.

The monerod binary is copied to $OUT which means we can use it on the host system. As such, this is the binary we send a lot of requests to.

In order to extract the code coverage HTML report, we need to have access to the source code we use for compilation of monerod. Specifically, to fuzz the monerod binary we launch it and use another process with a python-based random generator to send a number of requests to the binary. Once that is over, we gracefully terminated the monerod server, which will now hav produced a default.profraw file holding coverage information. To convert this into a HTML server we copy the default.profraw file into Monero's OSS-Fuzz docker image and run llvm-profdata merge ...; llvm-cov show html ... to create a HTML report. This is specifically achieved with the following lines in the e2e.py script:

```
# Define the command script that will be run inside the container
script = ('cd $SRC/monero/monero && '
          'git submodule init && '
          'git submodule update && '
          'llvm-profdata merge -sparse $(find /data -name "*.profraw") '
          '-o /data/monerod.profdata && '
```

```
            'llvm-cov show '
            '-format=html '
            '-output-dir=/data/coverage '
            '-Xdemangler c++filt '
            '-instr-profile=/data/monerod.profdata '
            '/data/monerod && '
            f'chown -R {uid}:{gid} /data/coverage')

# Run coverage generation in docker image
command = [
    'docker', 'run', '--rm', '-it', '-v', f'{workdir}:/data',
    'gcr.io/oss-fuzz/monero-rpc', 'bash', '-c', script
]

subprocess.check_call(command)
```

The fuzzing loop itself is simply a Python module that starts the relevant `monerod` binary and proceeds to send a lot of arbitrary requests.

The important pieces of the code are, in `e2e.py`:

```
....
def main():
    ...
        monerod_path = build_end_to_end_setup(os.path.abspath(args.oss_fuzz),
                                              abs_workdir, args.proj)

    # Launch the monero server and start fuzzing.
    monerod_proc, log_file = start_monerod(monerod_path, abs_workdir, 0)

    # Perform the actual fuzzing.
    rpc_call_stats = e2e_fuzzer.fuzz(args.round, abs_workdir, args.debug,
                                     rpc_call_stats, args.duration)

    # Ensure monerod is stopped
    stop_monerod(monerod_proc, log_file)

    # Dumping functions called count.
    dump_called_functions(abs_workdir, rpc_call_stats)

    # Process coverage report
...
```

and the important pieces of `e2e_fuzzer.fuzz` are:

```
def fuzz(max_rpc_requests_to_send: int, workdir: str, need_debug: bool,
         rpc_call_stats: dict[str, tuple[int, int]],
         duration: int) -> dict[str, tuple[int, int]]:
    """Launch a fuzzing campaign for the Monero RPC endpoints."""
    print('Fuzzing launching with max of %d rpc requests.' %
          max_rpc_requests_to_send)
...

    for rpc_request_counter in range(max(max_rpc_requests_to_send, 1)):
        ...
        # rpc_call_to_do = send_getblocktemplate# rpc_calls[rpc_index]
        rpc_call_to_do = rpc_calls[rpc_index]
        request, endpoint = rpc_call_to_do()
        if isinstance(request, bytes):
            success, _ = send_bin_request(request, endpoint)
        else:
            success, _ = send_request(request, endpoint)
...
```

Each entry in the `rpc_calls` list is a set of functions that generate random requests for a given RPF endpoint. There is a single function per request handler that takes care of generating random input and sending the request. For example, for `calc_pow` RPC call, we use the fuzz function:

```python
def send_calc_pow():
    request = generate_request(method='calc_pow',
                               params={
                                   'major_version': gen_random_int(0, 255),
                                   'height': gen_random_int(0, get_height()),
                                   'block_blob': gen_random_blob(),
                                   'seed_hash': gen_random_hex_string(64),
                               })
    return request, 'json_rpc'
```

where `generate_request` is defined as follows:

```python
def generate_request(method, params):
    request = {'jsonrpc': '2.0', 'id': '1', 'method': method, 'params': params}
    return request
```

The `send_request` function then simply sends the request on the network:

```python
def send_request(request, endpoint) -> tuple[bool, str]:
    ...

    # Fuzz the chosen target
    if debug:
        print('-----------------------------------')
        print('Sending to endpoint: %s' % endpoint)
        print('Parameters: %s ' % json.dumps(request))
    try:
        x = requests.post('http://127.0.0.1:38081/%s' % (endpoint),
                          json=request,
                          timeout=30)
        if debug:
            print('Response: %s ' % x.text)
        return True, x.text
    except requests.exceptions.Timeout:
        ...
```

5.2 Evaluation and performance

In order to run the end-to-end harness the following steps are necessary:

```
git clone https://github.com/google/oss-fuzz
cp -rf ADA_MONERO_DIR/monero-rpc oss-fuzz/projects/monero-rpc
python3 ADA_MONERO_DIR/e2e-testing/e2e.py --oss-fuzz ./oss-fuzz \
                                          --debug \
                                          --round 200 \
                                          --project monero-rpc \
                                          --workdir ./result
```

The `e2e.py` command above will:

1) Build `monerod` with the relevant sanitizeirs
2) Start `monerod`
3) Launch the python end-to-end fuzzing
4) extract code coverage

The output of the above will include a lot of debugging statements, which are good for getting an intuition of what is going on. You can disable this by removing `--debug`.

At the end of the `e2e.py` command you will get an output showing the location of the coverage report:

```
...
Submodule path 'external/randomx': checked out '102
    f8acf90a7649ada410de5499a7ec62e49e1da'
Submodule path 'external/rapidjson': checked out '129
    d19ba7f496df5e33658527a7158c79b99c21c'
Submodule path 'external/supercop': checked out '633500
    ad8c8759995049ccd022107d1fa8a1bbc9'
error: /src/monero/monero/build2/generated_include/crypto/wallet/ops.h: No such file or
     directory
warning: The file '/src/monero/monero/build2/generated_include/crypto/wallet/ops.h' isn
    't covered.
error: /src/monero/monero/build2/translations/translation_files.h: No such file or
    directory
warning: The file '/src/monero/monero/build2/translations/translation_files.h' isn't
    covered.
Coverage report available at: path_on_your_system/result/coverage
```

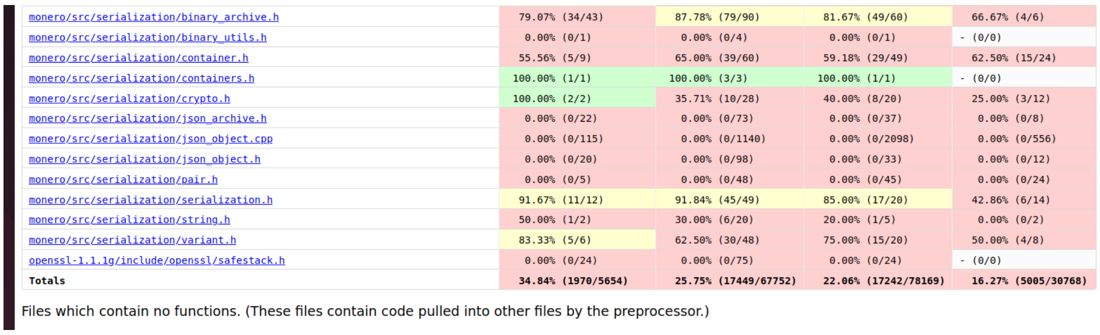
The coverage report can then be visualized with:

```
python3 -m http.server 8013 --directory path_on_your_system/result/coverage
```

You can adjust the harness runtime by either changing `round`, which is the number of RPC requests to send, or using `--duration XX` where `XX` is the number of seconds to run the python end-to-end fuzzing.

At the time of writing, from a run of 2000 random RPC requests, the coverage is as follows (the numbers are, in order, Function Coverage, Line Coverage, Region Coverage, Branch Coverage):

The total coverage

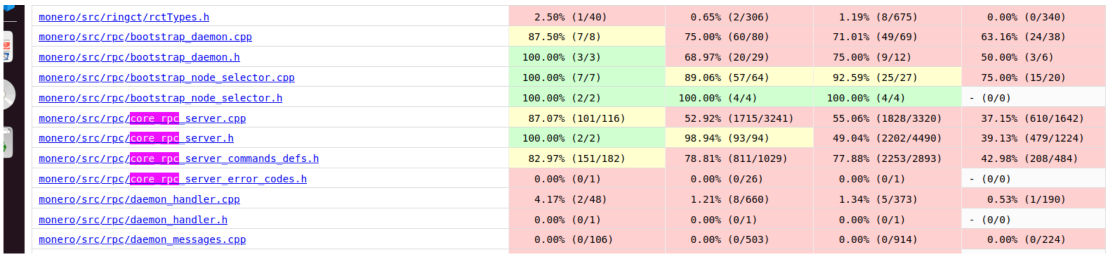


monero/src/serialization/binary_archive.h	79.07% (34/43)	87.78% (79/90)	81.67% (49/60)	66.67% (4/6)
monero/src/serialization/binary_utils.h	0.00% (0/1)	0.00% (0/4)	0.00% (0/1)	- (0/0)
monero/src/serialization/container.h	55.56% (5/9)	65.00% (39/60)	59.18% (29/49)	62.50% (15/24)
monero/src/serialization/containers.h	100.00% (1/1)	100.00% (3/3)	100.00% (1/1)	- (0/0)
monero/src/serialization/crypto.h	100.00% (2/2)	35.71% (10/28)	40.00% (8/20)	25.00% (3/12)
monero/src/serialization/json_archive.h	0.00% (0/22)	0.00% (0/73)	0.00% (0/37)	0.00% (0/8)
monero/src/serialization/json_object.cpp	0.00% (0/115)	0.00% (0/1140)	0.00% (0/2098)	0.00% (0/556)
monero/src/serialization/json_object.h	0.00% (0/20)	0.00% (0/98)	0.00% (0/33)	0.00% (0/12)
monero/src/serialization/pair.h	0.00% (0/5)	0.00% (0/48)	0.00% (0/45)	0.00% (0/24)
monero/src/serialization/serialization.h	91.67% (11/12)	91.84% (45/49)	85.00% (17/20)	42.86% (6/14)
monero/src/serialization/string.h	50.00% (1/2)	30.00% (6/20)	20.00% (1/5)	0.00% (0/2)
monero/src/serialization/variant.h	83.33% (5/6)	62.50% (30/48)	75.00% (15/20)	50.00% (4/8)
openssl-1.1.1g/include/openssl/safestack.h	0.00% (0/24)	0.00% (0/75)	0.00% (0/24)	- (0/0)
Totals	**34.84% (1970/5654)**	**25.75% (17449/67752)**	**22.06% (17242/78169)**	**16.27% (5005/30768)**

Files which contain no functions. (These files contain code pulled into other files by the preprocessor.)

A total of 1970 functions are covered in `monerod` during the fuzzing session and a total of aroud 26% line coverage. In cmoparison, the existing OSS-Fuzz set up has a coverage of 941 functions and 10192 lines covered, as per 15th August 2025 (coverage report).

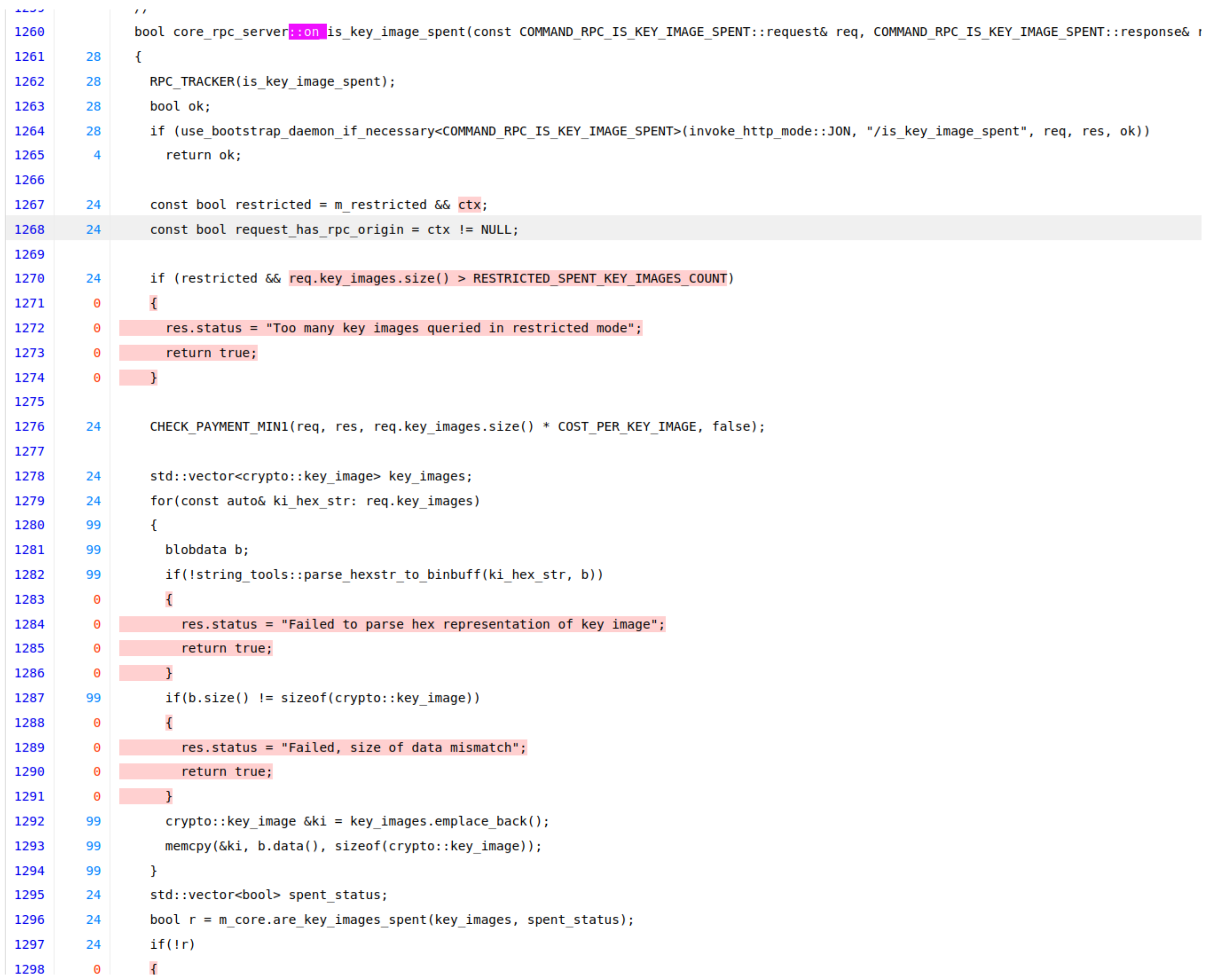
And the specific RPC coverage is:



monero/src/ringct/rctTypes.h	2.50% (1/40)	0.65% (2/306)	1.19% (8/675)	0.00% (0/340)
monero/src/rpc/bootstrap_daemon.cpp	87.50% (7/8)	75.00% (60/80)	71.01% (49/69)	63.16% (24/38)
monero/src/rpc/bootstrap_daemon.h	100.00% (3/3)	68.97% (20/29)	75.00% (9/12)	50.00% (3/6)
monero/src/rpc/bootstrap_node_selector.cpp	100.00% (7/7)	89.06% (57/64)	92.59% (25/27)	75.00% (15/20)
monero/src/rpc/bootstrap_node_selector.h	100.00% (2/2)	100.00% (4/4)	100.00% (4/4)	- (0/0)
monero/src/rpc/core_rpc_server.cpp	87.07% (101/116)	52.92% (1715/3241)	55.06% (1828/3320)	37.15% (610/1642)
monero/src/rpc/core_rpc_server.h	100.00% (2/2)	98.94% (93/94)	49.04% (2202/4490)	39.13% (479/1224)
monero/src/rpc/core_rpc_server_commands_defs.h	82.97% (151/182)	78.81% (811/1029)	77.88% (2253/2893)	42.98% (208/484)
monero/src/rpc/core_rpc_server_error_codes.h	0.00% (0/1)	0.00% (0/26)	0.00% (0/1)	- (0/0)
monero/src/rpc/daemon_handler.cpp	4.17% (2/48)	1.21% (8/660)	1.34% (5/373)	0.53% (1/190)
monero/src/rpc/daemon_handler.h	0.00% (0/1)	0.00% (0/1)	0.00% (0/1)	- (0/0)
monero/src/rpc/daemon_messages.cpp	0.00% (0/106)	0.00% (0/503)	0.00% (0/914)	0.00% (0/224)

A total of 101 functions of 116 functions in `core_rpc_server.cpp` are covered and a line coverage count of roughly 53%.

The RPC handlers have a lot of conditionals and various branches, and considering the harness does not rely on coverage feedback it's less deterministic as to how fast it explores the code. Our practice has been to have this fuzzer harness run routinely on a daily basis, and a sample of a RPC handler's coverage is as follows:



```
        //
        bool core_rpc_server::on_is_key_image_spent(const COMMAND_RPC_IS_KEY_IMAGE_SPENT::request& req, COMMAND_RPC_IS_KEY_IMAGE_SPENT::response& r
  28    {
  28      RPC_TRACKER(is_key_image_spent);
  28      bool ok;
  28      if (use_bootstrap_daemon_if_necessary<COMMAND_RPC_IS_KEY_IMAGE_SPENT>(invoke_http_mode::JON, "/is_key_image_spent", req, res, ok))
   4        return ok;

  24      const bool restricted = m_restricted && ctx;
  24      const bool request_has_rpc_origin = ctx != NULL;

  24      if (restricted && req.key_images.size() > RESTRICTED_SPENT_KEY_IMAGES_COUNT)
   0      {
   0        res.status = "Too many key images queried in restricted mode";
   0        return true;
   0      }

  24      CHECK_PAYMENT_MIN1(req, res, req.key_images.size() * COST_PER_KEY_IMAGE, false);

  24      std::vector<crypto::key_image> key_images;
  24      for(const auto& ki_hex_str: req.key_images)
  99      {
  99        blobdata b;
  99        if(!string_tools::parse_hexstr_to_binbuff(ki_hex_str, b))
   0        {
   0          res.status = "Failed to parse hex representation of key image";
   0          return true;
   0        }
  99        if(b.size() != sizeof(crypto::key_image))
   0        {
   0          res.status = "Failed, size of data mismatch";
   0          return true;
   0        }
  99        crypto::key_image &ki = key_images.emplace_back();
  99        memcpy(&ki, b.data(), sizeof(crypto::key_image));
  99      }
  24      std::vector<bool> spent_status;
  24      bool r = m_core.are_key_images_spent(key_images, spent_status);
  24      if(!r)
   0      {
```

6 Conclusions

In this report we have gone through the primary contributions of our Monero fuzzing audit. The central goal was to enable fuzzing of Monero's RPC handlers and do so by way of OSS-Fuzz. This has been achieved and there is at the time of writing an active pull request available in Monero's repository which holds the harness for this. The fuzzer relies on the fuzz data to construct a sequence of RPC calls to make, including fuzzer-seeded input to each RPC handler. The harness was developed through several iterations of the current version of the harness has run for 4 consecutive days without running into any issues and continuously increasing coverage exploration.

In addition to the libfuzzer harness, we developed an end-to-end style harness that relies on out-of-process fuzzing using the `monerod` binary directly. This is based on a Python fuzzer that sends arbitrary RPC requests over the local network, and also the Monero's OSS-Fuzz project to build a sanitized version of `monerod` both with Address Sanitizer and coverage sanitizer. As a result we're also able to extract ASAN reports for issues with this fuzzer and generate HTML coverage reports. This fuzzer can run with a fully synced blockchain, and can essentially work with any configurations of `monerod` as such. This makes it versatile and is easy to run in a manual way, to e.g. test new functionality or alike.

As the upstream Monero PR is still waiting to be merged into the repository, we expect to continue to monitor this until the fuzzer runs in OSS-Fuzz. Furthermore, once the fuzzer starts running in OSS-Fuzz then we consider it likely that further issues will be reported, and we are happy to commit to monitoring the running of the fuzzer and perform adjustments in case these are needed.