boofuzz Documentation

Release 0.2.0

Joshua Pereyda

Jul 22, 2020

USER GUIDE

Boofuzz is a fork of and the successor to the venerable Sulley fuzzing framework. Besides numerous bug fixes, boofuzz aims for extensibility. The goal: fuzz everything.

CHAPTER

ONE

WHY?

Sulley has been the preeminent open source fuzzer for some time, but has fallen out of maintenance.

CHAPTER

TWO

FEATURES

Like Sulley, boofuzz incorporates all the critical elements of a fuzzer:

- Easy and quick data generation.
- Instrumentation – AKA failure detection.
- Target reset after failure.
- Recording of test data.

Unlike Sulley, boofuzz also features:

- Much easier install experience!
- Support for arbitrary communications mediums.
- Built-in support for serial fuzzing, ethernet- and IP-layer, UDP broadcast.
- Better recording of test data – consistent, thorough, clear.
- Test result CSV export.
- *Extensible* instrumentation/failure detection.
- Far fewer bugs.

Sulley is affectionately named after the giant teal and purple creature from Monsters Inc. due to his fuzziness. Boofuzz is likewise named after the only creature known to have scared Sulley himself: Boo!

Fig. 1: Boo from Monsters Inc

CHAPTER

THREE

INSTALLATION

```
pip install boofuzz
```

Boofuzz installs as a Python library used to build fuzzer scripts. See *Installing boofuzz* for advanced and detailed instructions.

3.1 Installing boofuzz

3.1.1 Prerequisites

Boofuzz requires Python 2.7 or 3.5. Recommended installation requires `pip`. To ensure forward compatibility, Python 3 is recommended. As a base requirement, the following packages are needed:

Ubuntu/Debian `sudo apt-get install python3-pip python3-venv build-essential`

OpenSuse `sudo zypper install python3-devel gcc`

CentOS `sudo yum install python3-devel gcc`

3.1.2 Install

It is strongly recommended to set up boofuzz in a virtual environment (venv). However, the `venv` module is only available for Python 3. For Python 2.7, please use the older virtualenv package. First, create a directory that will hold our boofuzz install:

```
$ mkdir boofuzz && cd boofuzz
$ python3 -m venv env
```

This creates a new virtual environment env in the current folder. Note that the Python version in a virtual environment is fixed and chosen at its creation. Unlike global installs, within a virtual environment `python` is aliased to the Python version of the virtual environment.

Next, activate the virtual environment:

```
$ source env/bin/activate
```

Or, if you are on Windows:

```
> env\Scripts\activate.bat
```

Ensure you have the latest version of both `pip` and `setuptools`:

```
(env) $ pip install -U pip setuptools
```

Finally, install boofuzz:

```
(env) $ pip install boofuzz
```

To run and test your fuzzing scripts, make sure to always activate the virtual environment beforehand.

3.1.3 From Source

1. Like above, it is recommended to set up a virtual environment. Depending on your concrete setup, this is largely equivalent to the steps outlined above. Make sure to upgrade `setuptools` and `pip`.
2. Download the source code. You can either grab a zip from https://github.com/jtpereyda/boofuzz or directly clone it with git:

   ```
   $ git clone https://github.com/jtpereyda/boofuzz.git
   ```

3. Install. Run `pip` from within the boofuzz directory after activating the virtual environment:

   ```
   $ pip install .
   ```

Tips:

- Use the `-e` option for developer mode, which allows changes to be seen automatically without reinstalling:

  ```
  $ pip install -e .
  ```

- To install developer tools (unit test dependencies, test runners, etc.) as well:

  ```
  $ pip install -e .[dev]
  ```

  Note that black needs Python 3.6.

- If you're behind a proxy:

  ```
  $ set HTTPS_PROXY=http://your.proxy.com:port
  ```

- If you're planning on developing boofuzz itself, you can save a directory and create your virtual environment after you've cloned the source code (so `env/` is within the main boofuzz directory).

3.1.4 Extras

process_monitor.py (Windows only)

Warning: Currently, the process monitor is Python 2 only due to a dependency on `pydbg`. See the discussion in Issue #370 for more information regarding Python 3 support.

As always, contributions are welcome!

The process monitor is a tool for detecting crashes and restarting an application on Windows (process_monitor_unix.py is provided for Unix).

The process monitor is included with boofuzz, but requires additional libraries to run. While boofuzz typically runs on a different machine than the target, the process monitor must run on the target machine itself.

If you want to use process_monitor.py, follow these additional steps:

1. Download and install pydbg.
    1. Make sure to install and run pydbg using a 32-bit Python 2 interpreter, not 64-bit!
    2. The OpenRCE repository doesn't have a setup.py. Use Fitblip's fork.
    3. `C:\Users\IEUser\Downloads\pydbg>pip install .`
2. Download and install pydasm.
    1. `C:\Users\IEUser\Downloads\libdasm\pydasm>python setup.py build_ext**`
    2. `C:\Users\IEUser\Downloads\libdasm\pydasm>python setup.py install`
3. Verify that process_monitor.py runs:

```
C:\Users\IEUser\Downloads\boofuzz>python process_monitor.py -h
usage: procmon [-h] [--debug] [--quiet] [-f STR] [-c FILENAME] [-i PID]
               [-l LEVEL] [-p NAME] [-P PORT]

optional arguments:
  -h, --help            show this help message and exit
  --debug               toggle debug output
  --quiet               suppress all output
  -f STR, --foo STR     the notorious foo option
  -c FILENAME, --crash_bin FILENAME
                        filename to serialize crash bin class to
  -i PID, --ignore_pid PID
                        PID to ignore when searching for target process
  -l LEVEL, --log_level LEVEL
                        log level: default 1, increase for more verbosity
  -p NAME, --proc_name NAME
                        process name to search for and attach to
  -P PORT, --port PORT  TCP port to bind this agent to
```

** Building pydasm on Windows requires the Visual C++ Compiler for Python 2.7.

network_monitor.py

The network monitor was Sulley's primary tool for recording test data, and has been replaced with boofuzz's logging mechanisms. However, some people still prefer the PCAP approach.

Note: The network monitor requires Pcapy, which will not be automatically installed with boofuzz. You can manually install it with `pip install pcapy`.

If you run into errors, check out the requirements on the project page.

3.2 Quickstart

The *Session* object is the center of your fuzz... session. When you create it, you'll pass it a *Target* object, which will itself receive a *Connection* object. For example:

```
session = Session(
    target=Target(
        connection=TCPSocketConnection("127.0.0.1", 8021)))
```

Connection objects implement *ITargetConnection*. Available options include *TCPSocketConnection* and its sister classes for UDP, SSL and raw sockets, and *SerialConnection*.

With a Session object ready, you next need to define the messages in your protocol. Once you've read the requisite RFC, tutorial, etc., you should be confident enough in the format to define your protocol using the various *static protocol definition functions*.

Each message starts with an *s_initialize* function.

Here are several message definitions from the FTP protocol:

```
s_initialize("user")
s_string("USER")
s_delim(" ")
s_string("anonymous")
s_static("\r\n")

s_initialize("pass")
s_string("PASS")
s_delim(" ")
s_string("james")
s_static("\r\n")

s_initialize("stor")
s_string("STOR")
s_delim(" ")
s_string("AAAA")
s_static("\r\n")

s_initialize("retr")
s_string("RETR")
s_delim(" ")
s_string("AAAA")
s_static("\r\n")
```

Once you've defined your message(s), you will connect them into a graph using the Session object you just created.

```
session.connect(s_get("user"))
session.connect(s_get("user"), s_get("pass"))
session.connect(s_get("pass"), s_get("stor"))
session.connect(s_get("pass"), s_get("retr"))
```

After that, you are ready to fuzz:

```
session.fuzz()
```

Note that at this point you have only a very basic fuzzer. Making it kick butt is up to you. There are some examples and request_definitions in the repository that might help you get started.

The log data of each run will be saved to a SQLite database located in the **boofuzz-results** directory at your current workdir. You can reopen the webinterface on any of those databases at any time with

```
$ boo open <run-*.db>
```

To do cool stuff like checking responses, you'll want to use `post_test_case_callbacks` in *`Session`*. You may also be interested in *Making Your Own Block/Primitive*.

Remember boofuzz is all Python, so everything is there for your customization. If you are doing crazy cool stuff, check out the *community info* and consider contributing back!

Happy fuzzing, and Godspeed!

3.3 Contributing

3.3.1 Issues and Bugs

If you have a bug report or idea for improvement, please create an issue on GitHub, or a pull request with the fix.

3.3.2 Code Reviews

All pull requests are subject to professional code review. If you do not want your code reviewed, do not submit it.

3.3.3 Contributors

See installation instructions for details on installing boofuzz from source with developer options.

Pull Request Checklist

1. Install python version 2.7.9+ **and** 3.6+
2. Verify tests pass:

   ```
   tox
   ```

   Note: (Re-)creating a tox environment on Linux requires root rights because some of your unit tests work with raw sockets. tox will check if `cap_net_admin` and `cap_net_raw+eip` are set on the tox environment python interpreter and if not, will do so.

   Once the capabilities have been set, running tox won't need extended permissions.

   Attention: If the tests pass, check the output for new flake8 warnings that indicate PEP8 violations.

3. Format the code to meet our code style requirements (needs python 3.6+):

   ```
   black .
   ```

   Use `# fmt: off` and `# fmt: on` around a block to disable formatting locally.

4. If you have PyCharm, use it to see if your changes introduce any new static analysis warnings.
5. Modify CHANGELOG.rst to say what you changed.
6. If adding a new module, consider adding it to the Sphinx docs (see `docs` folder).

3.3.4 Maintainers

Review Checklist

On every pull request:

1. Verify changes are sensible and in line with project goals.
2. Verify tests pass (continuous integration is OK for this).
3. Use PyCharm to check static analysis if changes are significant or non-trivial.
4. Verify CHANGELOG.rst is updated.
5. Merge in.

Release Checklist

Releases are deployed from GitHub Actions when a new release is created on GitHub.

Prep

1. Create release branch.
2. Increment version number from last release according to PEP 0440 and roughly according to the Semantic Versioning guidelines.
    1. In `boofuzz/__init__.py`.
    2. In `docs/conf.py`.
3. Modify CHANGELOG file for publication if needed.
4. Merge release branch.

Release

1. Create release on Github.
2. Verify GitHub Actions deployment succeeds.

CHAPTER

FOUR

PUBLIC PROTOCOL LIBRARIES

The following protocol libraries are free and open source, but the implementations are not at all close to full protocol coverage:

- boofuzz-ftp
- boofuzz-http

If you have an open source boofuzz protocol suite to share, please *let us know*!

4.1 Session

class boofuzz.**Session**(*session_filename=None*, *index_start=1*, *index_end=None*, *sleep_time=0.0*, *restart_interval=0*, *web_port=26000*, *keep_web_open=True*, *console_gui=False*, *crash_threshold_request=12*, *crash_threshold_element=3*, *restart_sleep_time=5*, *restart_callbacks=None*, *restart_threshold=None*, *restart_timeout=None*, *pre_send_callbacks=None*, *post_test_case_callbacks=None*, *post_start_target_callbacks=None*, *fuzz_loggers=None*, *fuzz_db_keep_only_n_pass_cases=0*, *receive_data_after_each_request=True*, *check_data_received_each_request=False*, *receive_data_after_fuzz=False*, *ignore_connection_reset=False*, *ignore_connection_aborted=False*, *ignore_connection_issues_when_sending_fuzz_data=True*, *ignore_connection_ssl_errors=False*, *reuse_target_connection=False*, *target=None*)

Bases: boofuzz.pgraph.graph.Graph

Extends pgraph.graph and provides a container for architecting protocol dialogs.

Parameters

- **session_filename** (*str*) – Filename to serialize persistent data to. Default None.
- **index_start** (*int*) –
- **index_end** (*int*) –
- **sleep_time** (*float*) – Time in seconds to sleep in between tests. Default 0.
- **restart_interval** (*int*) – Restart the target after n test cases, disable by setting to 0 (default).
- **console_gui** (*bool*) – Use curses to generate a static console screen similar to the webinterface. Has not been tested under Windows. Default False.
- **crash_threshold_request** (*int*) – Maximum number of crashes allowed before a request is exhausted. Default 12.

- **crash_threshold_element** (*int*) – Maximum number of crashes allowed before an element is exhausted. Default 3.
- **restart_sleep_time** (*int*) – Time in seconds to sleep when target can't be restarted. Default 5.
- **restart_callbacks** (*list of method*) – The registered method will be called after a failed post_test_case_callback Default None.
- **restart_threshold** (*int*) – Maximum number of retries on lost target connection. Default None (indefinitely).
- **restart_timeout** (*float*) – Time in seconds for that a connection attempt should be retried. Default None (indefinitely).
- **pre_send_callbacks** (*list of method*) – The registered method will be called prior to each fuzz request. Default None.
- **post_test_case_callbacks** (*list of method*) – The registered method will be called after each fuzz test case. Default None.
- **post_start_target_callbacks** (*list of method*) – Method(s) will be called after the target is started or restarted, say, by a process monitor.
- **web_port** (*int*) – Port for monitoring fuzzing campaign via a web browser. Default 26000.
- **keep_web_open** (*bool*) – Keep the webinterface open after session completion. Default True.
- **fuzz_loggers** (*list of ifuzz_logger.IFuzzLogger*) – For saving test data and results.. Default Log to STDOUT.
- **fuzz_db_keep_only_n_pass_cases** (*int*) – Minimize disk usage by only saving passing test cases if they are in the n test cases preceding a failure or error. Set to 0 to save after every test case (high disk I/O!). Default 0.
- **receive_data_after_each_request** (*bool*) – If True, Session will attempt to receive a reply after transmitting each non-fuzzed node. Default True.
- **check_data_received_each_request** (*bool*) – If True, Session will verify that some data has been received after transmitting each non-fuzzed node, and if not, register a failure. If False, this check will not be performed. Default False. A receive attempt is still made unless receive_data_after_each_request is False.
- **receive_data_after_fuzz** (*bool*) – If True, Session will attempt to receive a reply after transmitting a fuzzed message. Default False.
- **ignore_connection_reset** (*bool*) – Log ECONNRESET errors ("Target connection reset") as "info" instead of failures.
- **ignore_connection_aborted** (*bool*) – Log ECONNABORTED errors as "info" instead of failures.
- **ignore_connection_issues_when_sending_fuzz_data** (*bool*) – Ignore fuzz data transmission failures. Default True. This is usually a helpful setting to enable, as targets may drop connections once a message is clearly invalid.
- **ignore_connection_ssl_errors** (*bool*) – Log SSL related errors as "info" instead of failures. Default False.
- **reuse_target_connection** (*bool*) – If True, only use one target connection instead of reconnecting each test case. Default False.

- **target** (Target) – Target for fuzz session. Target must be fully initialized. Default None.

add_node(*node*)
Add a pgraph node to the graph. We overload this routine to automatically generate and assign an ID whenever a node is added.

Parameters **node** (*pgraph.Node*) – Node to add to session graph

add_target(*target*)
Add a target to the session. Multiple targets can be added for parallel fuzzing.

Parameters **target** (Target) – Target to add to session

build_webapp_thread(*port=26000*)

connect(*src*, *dst=None*, *callback=None*)
Create a connection between the two requests (nodes) and register an optional callback to process in between transmissions of the source and destination request. Leverage this functionality to handle situations such as challenge response systems. The session class maintains a top level node that all initial requests must be connected to. Example:

```
sess = sessions.session()
sess.connect(sess.root, s_get("HTTP"))
```

If given only a single parameter, sess.connect() will default to attaching the supplied node to the root node. This is a convenient alias and is identical to the second line from the above example:

```
sess.connect(s_get("HTTP"))
```

If you register callback method, it must follow this prototype:

```
def callback(target, fuzz_data_logger, session, node, edge, *args, **kwargs)
```

Where node is the node about to be sent, edge is the last edge along the current fuzz path to "node", session is a pointer to the session instance which is useful for snagging data such as session.last_recv which contains the data returned from the last socket transmission and sock is the live socket. A callback is also useful in situations where, for example, the size of the next packet is specified in the first packet. As another example, if you need to fill in the dynamic IP address of the target register a callback that snags the IP from sock.getpeername()[0].

Parameters

- **src** (*str or Request (pgrah.Node)*) – Source request name or request node
- **dst** (*str or Request (pgrah.Node), optional*) – Destination request name or request node
- **callback** (*def, optional*) – Callback function to pass received data to between node xmits. Default None.

Returns The edge between the src and dst.

Return type pgraph.Edge

example_test_case_callback(*target*, *fuzz_data_logger*, *session*, **args*, ***kwargs*)
Example call signature for methods given to *register_post_test_case_callback()*.

Parameters

- **target** (Target) – Target with sock-like interface.

- **fuzz_data_logger** (*ifuzz_logger.IFuzzLogger*) – Allows logging of test checks and passes/failures. Provided with a test case and test step already opened.
- **session** (Session) – Session object calling post_send. Useful properties include last_send and last_recv.
- **args** – Implementations should include *args and **kwargs for forward-compatibility.
- **kwargs** – Implementations should include *args and **kwargs for forward-compatibility.

export_file()
Dump various object values to disk.

@see: import_file()

feature_check()
Check all messages/features.

Returns None

fuzz()
Fuzz the entire protocol tree.

Iterates through and fuzzes all fuzz cases, skipping according to self.skip and restarting based on self.restart_interval.

If you want the web server to be available, your program must persist after calling this method. helpers.pause_for_signal() is available to this end.

Returns None

fuzz_by_name(*name*)
Fuzz a particular test case or node by name.

Parameters **name** (*str*) – Name of node.

fuzz_single_case(*mutant_index*)
Fuzz a test case by mutant_index.

Parameters **mutant_index** (*int*) – Positive non-zero integer.

Returns None

Raises **sex.SulleyRuntimeError** – If any error is encountered while executing the test case.

fuzz_single_node_by_path(*node_names*)
Fuzz a particular node via the path in node_names.

Parameters **node_names** (*list of str*) – List of node names leading to target.

import_file()
Load various object values from disk.

@see: export_file()

property netmon_results

num_mutations(*this_node=None*, *path=()*)
Number of total mutations in the graph. The logic of this routine is identical to that of fuzz(). See fuzz() for inline comments. The member variable self.total_num_mutations is updated appropriately by this routine.

Parameters

- **this_node** (*request (node)*) – Current node that is being fuzzed. Default None.
- **path** (*list*) – Nodes along the path to the current one being fuzzed. Default [].

Returns Total number of mutations in this session.

Return type int

register_post_test_case_callback(*method*)
Register a post- test case method.

The registered method will be called after each fuzz test case.

Potential uses:

- Closing down a connection.
- Checking for expected responses.

The order of callback events is as follows:

```
pre_send() - req - callback ... req - callback - post-test-case-callback
```

Parameters **method** (*function*) – A method with the same parameters as `post_send()`

server_init()
Called by fuzz() to initialize variables, web interface, etc.

test_case_data(*index*)
Return test case data object (for use by web server)

Parameters **index** (*int*) – Test case index

Returns Test case data object

Return type DataTestCase

transmit_fuzz(*sock*, *node*, *edge*, *callback_data*)
Render and transmit a fuzzed node, process callbacks accordingly.

Parameters

- **sock** (*Target, optional*) – Socket-like object on which to transmit node
- **node** (*pgraph.node.node (Node), optional*) – Request/Node to transmit
- **edge** (*pgraph.edge.edge (pgraph.edge), optional*) – Edge along the current fuzz path from "node" to next node.
- **callback_data** (*bytes*) – Data from previous callback.

transmit_normal(*sock*, *node*, *edge*, *callback_data*)
Render and transmit a non-fuzzed node, process callbacks accordingly.

Parameters

- **sock** (*Target, optional*) – Socket-like object on which to transmit node
- **node** (*pgraph.node.node (Node), optional*) – Request/Node to transmit
- **edge** (*pgraph.edge.edge (pgraph.edge), optional*) – Edge along the current fuzz path from "node" to next node.
- **callback_data** (*bytes*) – Data from previous callback.

4.2 Target

class boofuzz.**Target**(*connection*, *monitors=None*, *monitor_alive=None*, *max_recv_bytes=10000*, *repeater=None*, ***kwargs*)

Bases: object

Target descriptor container.

Takes an ITargetConnection and wraps send/recv with appropriate FuzzDataLogger calls.

Encapsulates pedrpc connection logic.

Contains a logger which is configured by Session.add_target().

Example

tcp_target = Target(SocketConnection(host='127.0.0.1', port=17971))

Parameters

- **connection** (*itarget_connection.ITargetConnection*) – Connection to system under test.
- **monitors** (*List[Union[IMonitor, pedrpc.Client]]*) – List of Monitors for this Target.
- **monitor_alive** – List of Functions that are called when a Monitor is alive. It is passed the monitor instance that became alive. Use it to e.g. set options on restart.
- **repeater** (repeater.Repeater) – Repeater to use for sending. Default None.

close()

Close connection to the target.

Returns None

monitors_alive()

Wait for the monitors to become alive / establish connection to the RPC server. This method is called on every restart of the target and when it's added to a session. After successful probing, a callback is called, passing the monitor.

Returns None

property netmon_options

open()

Opens connection to the target. Make sure to call close!

Returns None

pedrpc_connect()

property procmon_options

recv(*max_bytes=None*)

Receive up to max_bytes data from the target.

Parameters **max_bytes** (*int*) – Maximum number of bytes to receive.

Returns Received data.

send(*data*)

Send data to the target. Only valid after calling open!

Parameters **data** – Data to send.

Returns None

set_fuzz_data_logger(*fuzz_data_logger*)
Set this object's fuzz data logger – for sent and received fuzz data.

Parameters **fuzz_data_logger** (*ifuzz_logger.IFuzzLogger*) – New logger.

Returns None

4.2.1 Repeater

class boofuzz.repeater.**Repeater**(*sleep_time*)
Bases: object

Base Repeater class.

Parameters **sleep_time** (*float*) – Time to sleep between repetitions.

abstract log_message()
Formats a message to output in a log file. It should contain info about your repetition.

abstract repeat()
Decides whether the operation should repeat.

Returns True if the operation should repeat, False otherwise.

Return type Bool

abstract reset()
Resets the internal state of the repeater.

abstract start()
Starts the repeater.

The following concrete implementations of this interface are available:

4.2.2 TimeRepeater

class boofuzz.repeater.**TimeRepeater**(*duration*, *sleep_time=0*)
Bases: *boofuzz.repeater.Repeater*

Time-based repeater class. Starts a timer, and repeats until *duration* seconds have passed.

Raises **ValueError** – Raised if a time <= 0 is specified.

Parameters

- **duration** (*float*) – The duration of the repitition.
- **sleep_time** (*float*) – Time to sleep between repetitions.

log_message()
Formats a message to output in a log file. It should contain info about your repetition.

repeat()
Decides whether the operation should repeat.

Returns True if the operation should repeat, False otherwise.

Return type Bool

reset()
Resets the timer.

start()
Starts the timer.

4.2.3 CountRepeater

class boofuzz.repeater.**CountRepeater**(*count*, *sleep_time=0*)
Bases: *boofuzz.repeater.Repeater*

Count-Based repeater class. Repeats a fixed number of times.

Raises **ValueError** – Raised if a count < 1 is specified.

Parameters

- **count** (*int*) – Total amount of packets to be sent. **Important**: Do not confuse this parameter with the amount of repetitions. Specifying 1 would send exactly one packet.
- **sleep_time** (*float*) – Time to sleep between repetitions.

log_message()
Formats a message to output in a log file. It should contain info about your repetition.

repeat()
Decides whether the operation should repeat.

Returns True if the operation should repeat, False otherwise.

Return type Bool

reset()
Resets the internal state of the repeater.

start()
Starts the repeater.

4.3 Connections

Connection objects implement *ITargetConnection*. Available options include:

- *TCPSocketConnection*
- *UDPSocketConnection*
- *SSLSocketConnection*
- *RawL2SocketConnection*
- *RawL3SocketConnection*
- *SocketConnection (depreciated)*
- *SerialConnection*

4.3.1 ITargetConnection

class boofuzz.connections.**ITargetConnection**
Bases: object

Interface for connections to fuzzing targets. Target connections may be opened and closed multiple times. You must open before using send/recv and close afterwards.

Changed in version 0.2.0: ITargetConnection has been moved into the connections subpackage. The full path is now boofuzz.connections.itarget_connection.ITargetConnection

abstract close()
Close connection.

Returns None

abstract property info
Return description of connection info.

E.g., "127.0.0.1:2121"

Returns Connection info descrption

Return type str

abstract open()
Opens connection to the target. Make sure to call close!

Returns None

abstract recv(*max_bytes*)
Receive up to max_bytes data.

Parameters **max_bytes** (*int*) – Maximum number of bytes to receive.

Returns Received data. bytes('') if no data is received.

Return type bytes

abstract send(*data*)
Send data to the target.

Parameters **data** – Data to send.

Returns Number of bytes actually sent.

Return type int

4.3.2 BaseSocketConnection

class boofuzz.connections.**BaseSocketConnection**(*send_timeout*, *recv_timeout*)
Bases: boofuzz.connections.itarget_connection.ITargetConnection

This class serves as a base for a number of Connections over sockets.

New in version 0.2.0.

Parameters

- **send_timeout** (*float*) – Seconds to wait for send before timing out. Default 5.0.
- **recv_timeout** (*float*) – Seconds to wait for recv before timing out. Default 5.0.

close()
Close connection to the target.

Returns None

abstract open()
Opens connection to the target. Make sure to call close!

Returns None

4.3.3 TCPSocketConnection

class boofuzz.connections.**TCPSocketConnection**(*host*, *port*, *send_timeout=5.0*, *recv_timeout=5.0*, *server=False*)
Bases: boofuzz.connections.base_socket_connection.BaseSocketConnection

BaseSocketConnection implementation for use with TCP Sockets.

New in version 0.2.0.

Parameters

- **host** (*str*) – Hostname or IP adress of target system.
- **port** (*int*) – Port of target service.
- **send_timeout** (*float*) – Seconds to wait for send before timing out. Default 5.0.
- **recv_timeout** (*float*) – Seconds to wait for recv before timing out. Default 5.0.
- **server** (*bool*) – Set to True to enable server side fuzzing.

close()
Close connection to the target.

Returns None

property info
Return description of connection info.

E.g., "127.0.0.1:2121"

Returns Connection info descrption

Return type str

open()
Opens connection to the target. Make sure to call close!

Returns None

recv(*max_bytes*)
Receive up to max_bytes data from the target.

Parameters **max_bytes** (*int*) – Maximum number of bytes to receive.

Returns Received data.

send(*data*)
Send data to the target. Only valid after calling open!

Parameters **data** – Data to send.

Returns Number of bytes actually sent.

Return type int

4.3.4 UDPSocketConnection

class boofuzz.connections.**UDPSocketConnection**(*host*, *port*, *send_timeout=5.0*, *recv_timeout=5.0*, *server=False*, *bind=None*, *broadcast=False*)

Bases: boofuzz.connections.base_socket_connection.BaseSocketConnection

BaseSocketConnection implementation for use with UDP Sockets.

New in version 0.2.0.

Parameters

- **host** (*str*) – Hostname or IP adress of target system.
- **port** (*int*) – Port of target service.
- **send_timeout** (*float*) – Seconds to wait for send before timing out. Default 5.0.
- **recv_timeout** (*float*) – Seconds to wait for recv before timing out. Default 5.0.
- **server** (*bool*) – Set to True to enable server side fuzzing.
- **bind** (*tuple (host, port)*) – Socket bind address and port. Required if using recv().
- **broadcast** (*bool*) – Set to True to enable UDP broadcast. Must supply appropriate broadcast address for send() to work, and '' for bind host for recv() to work.

property info

Return description of connection info.

E.g., "127.0.0.1:2121"

Returns Connection info descrption

Return type str

classmethod max_payload()

Returns the maximum payload this connection can send at once.

This performs some crazy CTypes magic to do a getsockopt() which determines the max UDP payload size in a platform-agnostic way.

Returns The maximum length of a UDP packet the current platform supports

Return type int

open()

Opens connection to the target. Make sure to call close!

Returns None

recv(*max_bytes*)

Receive up to max_bytes data from the target.

Parameters **max_bytes** (*int*) – Maximum number of bytes to receive.

Returns Received data.

send(*data*)

Send data to the target. Only valid after calling open! Some protocols will truncate; see self.MAX_PAYLOADS.

Parameters **data** – Data to send.

Returns Number of bytes actually sent.

Return type int

4.3.5 SSLSocketConnection

class boofuzz.connections.**SSLSocketConnection**(*host*, *port*, *send_timeout=5.0*, *recv_timeout=5.0*, *server=False*, *sslcontext=None*, *server_hostname=None*)

Bases: boofuzz.connections.tcp_socket_connection.TCPSocketConnection

BaseSocketConnection implementation for use with SSL Sockets.

New in version 0.2.0.

Parameters

- **host** (*str*) – Hostname or IP adress of target system.
- **port** (*int*) – Port of target service.
- **send_timeout** (*float*) – Seconds to wait for send before timing out. Default 5.0.
- **recv_timeout** (*float*) – Seconds to wait for recv before timing out. Default 5.0.
- **server** (*bool*) – Set to True to enable server side fuzzing.
- **sslcontext** (*ssl.SSLContext*) – Python SSL context to be used. Required if server=True or server_hostname=None.
- **server_hostname** (*string*) – server_hostname, required for verifying identity of remote SSL/TLS server

open()

Opens connection to the target. Make sure to call close!

Returns None

recv(*max_bytes*)

Receive up to max_bytes data from the target.

Parameters **max_bytes** (*int*) – Maximum number of bytes to receive.

Returns Received data.

send(*data*)

Send data to the target. Only valid after calling open!

Parameters **data** – Data to send.

Returns Number of bytes actually sent.

Return type int

4.3.6 RawL2SocketConnection

class boofuzz.connections.**RawL2SocketConnection**(*interface*, *send_timeout=5.0*, *recv_timeout=5.0*, *ethernet_proto=0*, *mtu=1518*, *has_framecheck=True*)

Bases: boofuzz.connections.base_socket_connection.BaseSocketConnection

BaseSocketConnection implementation for use with Raw Layer 2 Sockets.

New in version 0.2.0.

Parameters

- **interface** (*str*) – Hostname or IP adress of target system.

- **send_timeout** (*float*) – Seconds to wait for send before timing out. Default 5.0.
- **recv_timeout** (*float*) – Seconds to wait for recv before timing out. Default 5.0.
- **ethernet_proto** (*int*) – Ethernet protocol to bind to. If supplied, the opened socket gets bound to this protocol, otherwise the python default of 0 is used. Must be supplied if this socket should be used for receiving. For valid options, see <net/if_ether.h> in the Linux Kernel documentation. Usually, ETH_P_ALL (0x0003) is not a good idea.
- **mtu** (*int*) – sets the maximum transmission unit size for this connection. Defaults to 1518 for standard Ethernet.
- **has_framecheck** (*bool*) – Indicates if the target ethernet protocol needs 4 bytes for a framecheck. Default True (for standard Ethernet).

property info

Return description of connection info.

E.g., "127.0.0.1:2121"

Returns Connection info descrption

Return type str

open()

Opens connection to the target. Make sure to call close!

Returns None

recv(*max_bytes*)

Receives a packet from the raw socket. If max_bytes < mtu, only the first max_bytes are returned and the rest of the packet is discarded. Otherwise, return the whole packet.

Parameters **max_bytes** (*int*) – Maximum number of bytes to return. 0 to return the whole packet.

Returns Received data

send(*data*)

Send data to the target. Only valid after calling open! Data will be trunctated to self.max_send_size (Default: 1514 bytes).

Parameters **data** – Data to send.

Returns Number of bytes actually sent.

Return type int

4.3.7 RawL3SocketConnection

class boofuzz.connections.**RawL3SocketConnection**(*interface*, *send_timeout=5.0*, *recv_timeout=5.0*, *ethernet_proto=2048*, *l2_dst=b'\xff\xff\xff\xff\xff\xff'*, *packet_size=1500*)

Bases: boofuzz.connections.base_socket_connection.BaseSocketConnection

BaseSocketConnection implementation for use with Raw Layer 2 Sockets.

New in version 0.2.0.

Parameters

- **interface** (*str*) – Interface to send and receive on.

- **send_timeout** (*float*) – Seconds to wait for send before timing out. Default 5.0.
- **recv_timeout** (*float*) – Seconds to wait for recv before timing out. Default 5.0.
- **ethernet_proto** (*int*) – Ethernet protocol to bind to. Defaults to ETH_P_IP (0x0800).
- **l2_dst** (*str*) – Layer2 destination address (e.g. MAC address). Default 'ÿÿÿÿÿÿ' (broadcast)
- **packet_size** (*int*) – Maximum packet size (in bytes). Default 1500 if the underlying interface uses standard ethernet for layer 2. Otherwise, a different packet size may apply (e.g. Jumboframes, 802.5 Token Ring, 802.11 wifi, ...) that must be specified.

property info

Return description of connection info.

E.g., "127.0.0.1:2121"

Returns Connection info descrption

Return type str

open()

Opens connection to the target. Make sure to call close!

Returns None

recv(*max_bytes*)

Receives a packet from the raw socket. If max_bytes < packet_size, only the first max_bytes are returned and the rest of the packet is discarded. Otherwise, return the whole packet.

Parameters **max_bytes** (*int*) – Maximum number of bytes to return. 0 to return the whole packet.

Returns Received data

send(*data*)

Send data to the target. Only valid after calling open! Data will be trunctated to self.packet_size (Default: 1500 bytes).

Parameters **data** – Data to send.

Returns Number of bytes actually sent.

Return type int

4.3.8 SocketConnection

boofuzz.connections.**SocketConnection**(*host*, *port=None*, *proto='tcp'*, *bind=None*, *send_timeout=5.0*, *recv_timeout=5.0*, *ethernet_proto=None*, *l2_dst=b'\xff\xff\xff\xff\xff\xff'*, *udp_broadcast=False*, *server=False*, *sslcontext=None*, *server_hostname=None*)

ITargetConnection implementation using sockets.

Supports UDP, TCP, SSL, raw layer 2 and raw layer 3 packets.

Note: SocketConnection is deprecated and will be removed in a future version of Boofuzz. Use the classes derived from *BaseSocketConnection* instead.

Changed in version 0.2.0: SocketConnection has been moved into the connections subpackage. The full path is now boofuzz.connections.socket_connection.SocketConnection

Deprecated since version 0.2.0: Use the classes derived from *BaseSocketConnection* instead.

Examples:

```
tcp_connection = SocketConnection(host='127.0.0.1', port=17971)
udp_connection = SocketConnection(host='127.0.0.1', port=17971, proto='udp')
udp_connection_2_way = SocketConnection(host='127.0.0.1', port=17971, proto='udp',
↪ bind=('127.0.0.1', 17972)
udp_broadcast = SocketConnection(host='127.0.0.1', port=17971, proto='udp', bind=(
↪'127.0.0.1', 17972),
                                 udp_broadcast=True)
raw_layer_2 = (host='lo', proto='raw-l2')
raw_layer_2 = (host='lo', proto='raw-l2',
               l2_dst='\xFF\xFF\xFF\xFF\xFF\xFF', ethernet_proto=socket_
↪connection.ETH_P_IP)
raw_layer_3 = (host='lo', proto='raw-l3')
```

Parameters

- **host** (*str*) – Hostname or IP address of target system, or network interface string if using raw-l2 or raw-l3.
- **port** (*int*) – Port of target service. Required for proto values 'tcp', 'udp', 'ssl'.
- **proto** (*str*) – Communication protocol ("tcp", "udp", "ssl", "raw-l2", "raw-l3"). Default "tcp". raw-l2: Send packets at layer 2. Must include link layer header (e.g. Ethernet frame). raw-l3: Send packets at layer 3. Must include network protocol header (e.g. IPv4).
- **bind** (*tuple (host, port)*) – Socket bind address and port. Required if using recv() with 'udp' protocol.
- **send_timeout** (*float*) – Seconds to wait for send before timing out. Default 5.0.
- **recv_timeout** (*float*) – Seconds to wait for recv before timing out. Default 5.0.
- **ethernet_proto** (*int*) – Ethernet protocol when using 'raw-l3'. 16 bit integer. Default ETH_P_IP (0x0800) when using 'raw-l3'. See "if_ether.h" in Linux documentation for more options.
- **l2_dst** (*str*) – Layer 2 destination address (e.g. MAC address). Used only by 'raw-l3'. Default 'ÿÿÿÿÿÿ' (broadcast).
- **udp_broadcast** (*bool*) – Set to True to enable UDP broadcast. Must supply appropriate broadcast address for send() to work, and '' for bind host for recv() to work.
- **server** (*bool*) – Set to True to enable server side fuzzing.
- **sslcontext** (*ssl.SSLContext*) – Python SSL context to be used. Required if server=True or server_hostname=None.
- **server_hostname** (*string*) – server_hostname, required for verifying identity of remote SSL/TLS server.

4.3.9 SerialConnection

class boofuzz.connections.**SerialConnection**(*port=0*, *baudrate=9600*, *timeout=5*, *message_separator_time=0.3*, *content_checker=None*)

Bases: boofuzz.connections.itarget_connection.ITargetConnection

ITargetConnection implementation for generic serial ports.

Since serial ports provide no default functionality for separating messages/packets, this class provides several means:

- timeout: Return received bytes after timeout seconds.
- msg_separator_time: Return received bytes after the wire is silent for a given time. This is useful, e.g., for terminal protocols without a machine-readable delimiter. A response may take a long time to send its information, and you know the message is done when data stops coming.
- content_check: A user-defined function takes the data received so far and checks for a packet. The function should return 0 if the packet isn't finished yet, or n if a valid message of n bytes has been received. Remaining bytes are stored for next call to recv(). Example:

```
def content_check_newline(data):
if data.find('\n') >= 0:
    return data.find('\n')
else:
    return 0
```

If none of these methods are used, your connection may hang forever.

Changed in version 0.2.0: SerialConnection has been moved into the connections subpackage. The full path is now boofuzz.connections.serial_connection.SerialConnection

Parameters

- **port** (*Union[int, str]*) – Serial port name or number.
- **baudrate** (*int*) – Baud rate for port.
- **timeout** (*float*) – For recv(). After timeout seconds from receive start, recv() will return all received data, if any.
- **message_separator_time** (*float*) – After message_separator_time seconds *without receiving any more data*, recv() will return. Optional. Default None.
- **content_checker** (*function(str) -> int*) – User-defined function. recv() will pass all bytes received so far to this method. If the method returns n > 0, recv() will return n bytes. If it returns 0, recv() will keep on reading.

close()

Close connection to the target.

Returns None

property info

Return description of connection info.

E.g., "127.0.0.1:2121"

Returns Connection info descrption

Return type str

open()
Opens connection to the target. Make sure to call close!

Returns None

recv(*max_bytes*)
Receive up to max_bytes data from the target.

Parameters **max_bytes** (*int*) – Maximum number of bytes to receive.

Returns Received data.

send(*data*)
Send data to the target. Only valid after calling open!

Parameters **data** – Data to send.

Returns Number of bytes actually sent.

Return type int

4.4 Monitors

Monitors are components that monitor the target for specific behaviour. A monitor can be passive and just observe and provide data or behave more actively, interacting directly with the target. Some monitors also have the capability to start, stop and restart targets.

Detecting a crash or misbehaviour of your target can be a complex, non-straight forward process depending on the tools you have available on your targets host; this holds true especially for embedded devices. Boofuzz provides three main monitor implementations:

- *ProcessMonitor*, a Monitor that collects debug info from process on Windows and Unix. It also can restart the target process and detect segfaults.
- *NetworkMonitor*, a Monitor that passively captures network traffic via PCAP and attaches it to the testcase log.
- *CallbackMonitor*, which is used to implement the callbacks that can be supplied to the Session class.

4.4.1 Monitor Interface (BaseMonitor)

class boofuzz.monitors.**BaseMonitor**
Bases: object

Interface for Target monitors. All Monitors must adhere to this specification.

New in version 0.2.0.

alive()
Called when a Target containing this Monitor is added to a session. Use this function to connect to e.g. RPC hosts if your target lives on another machine.

You MUST return True if the monitor is alive. You MUST return False otherwise. If a Monitor is not alive, this method will be called until it becomes alive or throws an exception. You SHOULD handle timeouts / connection retry limits in the monitor implementation.

Defaults to return True.

Returns Bool

get_crash_synopsis()
Called if any monitor indicates that the current testcase has failed, even if this monitor did not detect a crash. You SHOULD return a human- readable representation of the crash synopsis (e.g. hexdump). You MAY save the full crashdump somewhere.

Returns str

post_send(*target=None*, *fuzz_data_logger=None*, *session=None*)
Called after the current fuzz node is transmitted. Use it to collect data about a target and decide whether it crashed.

You MUST return True if the Target is still alive. You MUST return False if the Target crashed. If one Monitor reports a crash, the whole testcase will be marked as crashing.

Defaults to return True.

Returns Bool

post_start_target(*target=None*, *fuzz_data_logger=None*, *session=None*)
Called after a target is started or restarted.

pre_send(*target=None*, *fuzz_data_logger=None*, *session=None*)
Called before the current fuzz node is transmitted.

Defaults to no effect.

Returns None

restart_target(*target=None*, *fuzz_data_logger=None*, *session=None*)
Restart a target. Must return True if restart was successful, False if it was unsuccessful or this monitor cannot restart a Target, which causes the next monitor in the chain to try to restart.

The first successful monitor causes the restart chain to stop applying.

Defaults to call stop and start, return True if successful.

Returns Bool

retrieve_data()
Called to retrieve data independent of whether the current fuzz node crashed the target or not. Called before the fuzzer proceeds to a new testcase.

You SHOULD return any auxiliary data that should be recorded. The data MUST be serializable, e.g. bytestring.

Defaults to return None.

set_options(**args*, ***kwargs*)
Called to set options for your monitor (e.g. local crash dump storage). *args and **kwargs can be explicitly specified by implementing classes, however you SHOULD ignore any kwargs you do not recognize.

Defaults to no effect.

Returns None

start_target()
Starts a target. You MUST return True if the start was successful. You MUST return False if not. Monitors will be tried to start the target in the order they were added to the Target; the first Monitor to succeed breaks iterating.

Returns Bool

stop_target()
Stops a target. You MUST return True if the stop was successful. You MUST return False if not. Monitors

will be tried to stop the target in the order they were added to the Target; the first Monitor to succeed breaks iterating.

Returns Bool

4.4.2 ProcessMonitor

The process monitor consists of two parts; the `ProcessMonitor` class that implements `BaseMonitor` and a second module that is to be run on the host of your target.

class boofuzz.monitors.**ProcessMonitor**(*host*, *port*)

Proxy class for the process monitor interface.

In Versions < 0.2.0, boofuzz had network and process monitors that communicated over RPC. The RPC client was directly passed to the session class, and resolved all method calls dynamically on the RPC partner.

Since 0.2.0, every monitor class must implement the abstract class BaseMonitor, which defines a common interface among all Monitors. To aid future typehinting efforts and to disambiguate Network- and Process Monitors, this explicit proxy class has been introduced that fast-forwards all calls to the RPC partner.

New in version 0.2.0.

alive()

This method is forwarded to the RPC daemon.

get_crash_synopsis()

This method is forwarded to the RPC daemon.

on_new_server(*new_uuid*)

Restores all set options to the RPC daemon if it has restarted since the last call.

post_send(*target=None*, *fuzz_data_logger=None*, *session=None*)

This method is forwarded to the RPC daemon.

pre_send(*target=None*, *fuzz_data_logger=None*, *session=None*)

This method is forwarded to the RPC daemon.

restart_target(*target=None*, *fuzz_data_logger=None*, *session=None*)

This method is forwarded to the RPC daemon.

set_crash_filename(*new_crash_filename*)

Deprecated since version 0.2.0.

This option should be set via `set_options`.

set_options(**args*, ***kwargs*)

The old RPC interfaces specified set_foobar methods to set options. As these vary by RPC implementation, this trampoline method translates arguments that have been passed as keyword arguments to set_foobar calls.

If you call `set_options(foobar="barbaz")`, it will result in a call to `set_foobar("barbaz")` on the RPC partner.

set_proc_name(*new_proc_name*)

Deprecated since version 0.2.0.

This option should be set via `set_options`.

set_start_commands(*new_start_commands*)

Deprecated since version 0.2.0.

This option should be set via `set_options`.

set_stop_commands(*new_stop_commands*)
Deprecated since version 0.2.0.

This option should be set via `set_options`.

start_target()
This method is forwarded to the RPC daemon.

stop_target()
This method is forwarded to the RPC daemon.

4.4.3 NetworkMonitor

The network monitor consists of two parts; the `NetworkMonitor` class that implements `BaseMonitor` and a second module that is to be run on a host that can monitor the traffic.

class boofuzz.monitors.**NetworkMonitor**(*host*, *port*)
Proxy class for the network monitor interface.

In Versions < 0.2.0, boofuzz had network and process monitors that communicated over RPC. The RPC client was directly passed to the session class, and resolved all method calls dynamically on the RPC partner.

Since 0.2.0, every monitor class must implement the abstract class BaseMonitor, which defines a common interface among all Monitors. To aid future typehinting efforts and to disambiguate Network- and Process Monitors, this explicit proxy class has been introduced that fast-forwards all calls to the RPC partner.

New in version 0.2.0.

alive()
This method is forwarded to the RPC daemon.

on_new_server(*new_uuid*)
Restores all set options to the RPC daemon if it has restarted since the last call.

post_send(*target=None*, *fuzz_data_logger=None*, *session=None*)
This method is forwarded to the RPC daemon.

pre_send(*target=None*, *fuzz_data_logger=None*, *session=None*)
This method is forwarded to the RPC daemon.

restart_target(*target=None*, *fuzz_data_logger=None*, *session=None*)
Always returns false as this monitor cannot restart a target.

retrieve_data()
This method is forwarded to the RPC daemon.

set_filter(*new_filter*)
Deprecated since version 0.2.0.

This option should be set via `set_options`.

set_log_path(*new_log_path*)
Deprecated since version 0.2.0.

This option should be set via `set_options`.

set_options(**args*, ***kwargs*)
The old RPC interfaces specified set_foobar methods to set options. As these vary by RPC implementation, this trampoline method translates arguments that have been passed as keyword arguments to set_foobar calls.

If you call `set_options(foobar="barbaz")`, it will result in a call to `set_foobar("barbaz")` on the RPC partner.

Additionally, any options set here are cached and re-applied to the RPC server should it restart for whatever reason (e.g. the VM it's running on was restarted).

4.4.4 CallbackMonitor

class boofuzz.monitors.**CallbackMonitor**(*on_pre_send=None*, *on_post_send=None*, *on_restart_target=None*, *on_post_start_target=None*)

New-Style Callback monitor that is used in Session to provide callback-arrays. It's purpose is to keep the *_callbacks arguments in the session class while simplifying the implementation of session by forwarding these callbacks to the monitor infrastructure.

The mapping of arguments to method implementations of this class is as follows:

- restart_callbacks –> target_restart
- pre_send_callbacks –> pre_send
- post_test_case_callbacks –> post_send
- post_start_target_callbacks –> post_start_target

All other implemented interface members are stubs only, as no corresponding arguments exist in session. In any case, it is probably wiser to implement a custom Monitor than to use the callback functions.

New in version 0.2.0.

post_send(*target=None*, *fuzz_data_logger=None*, *session=None*)

This method iterates over all supplied post send callbacks and executes them. Their return values are discarded, exceptions are caught and logged:

- `BoofuzzTargetConnectionReset` will log a failure
- `BoofuzzTargetConnectionAborted` will log an info
- `BoofuzzTargetConnectionFailedError` will log a failure
- `BoofuzzSSLError` will log either info or failure, depending on if the session ignores SSL/TLS errors.
- every other exception is logged as an error.

All exceptions are discarded after handling.

post_start_target(*target=None*, *fuzz_data_logger=None*, *session=None*)

Called after a target is started or restarted.

pre_send(*target=None*, *fuzz_data_logger=None*, *session=None*)

This method iterates over all supplied pre send callbacks and executes them. Their return values are discarded, exceptions are catched and logged, but otherwise discarded.

restart_target(*target=None*, *fuzz_data_logger=None*, *session=None*)

This Method tries to restart a target. If no restart callbacks are set, it returns false; otherwise it returns true.

Returns bool

4.5 Logging

Boofuzz provides flexible logging. All logging classes implement *IFuzzLogger*. Built-in logging classes are detailed below.

To use multiple loggers at once, see *FuzzLogger*.

4.5.1 Logging Interface (IFuzzLogger)

class boofuzz.**IFuzzLogger**
Bases: object

Abstract class for logging fuzz data.

Usage while testing:

1. Open test case.
2. Open test step.
3. Use other log methods.

IFuzzLogger provides the logging interface for the Sulley framework and test writers.

The methods provided are meant to mirror functional test actions. Instead of generic debug/info/warning methods, IFuzzLogger provides a means for logging test cases, passes, failures, test steps, etc.

This hypothetical sample output gives an idea of how the logger should be used:

Test Case: UDP.Header.Address 3300

Test Step: Fuzzing Send: 45 00 13 ab 00 01 40 00 40 11 c9 . . .

Test Step: Process monitor check Check OK

Test Step: DNP Check Send: ff ff ff ff ff ff 00 0c 29 d1 10 . . . Recv: 00 0c 29 d1 10 81 00 30 a7 05 6e . . . Check: Reply is as expected. Check OK

Test Case: UDP.Header.Address 3301

Test Step: Fuzzing Send: 45 00 13 ab 00 01 40 00 40 11 c9 . . .

Test Step: Process monitor check Check Failed: "Process returned exit code 1"

Test Step: DNP Check Send: ff ff ff ff ff ff 00 0c 29 d1 10 . . . Recv: None Check: Reply is as expected. Check Failed

A test case is opened for each fuzzing case. A test step is opened for each high-level test step. Test steps can include, for example:

- Fuzzing
- Set up (pre-fuzzing)
- Post-test cleanup
- Instrumentation checks
- Reset due to failure

Within a test step, a test may log data sent, data received, checks, check results, and other information.

abstract close_test()
Called after a test has been completed. Can be used to inform the operator or save the test log.

Param None

Type None

Returns None

Return type None

abstract close_test_case()
Called after a test case has been completed. Can be used to inform the operator or save the test case log.

Param None

Type None

Returns None

Return type None

abstract log_check(*description*)
Records a check on the system under test. AKA "instrumentation check."

Parameters **description** (*str*) – Received data.

Returns None

Return type None

abstract log_error(*description*)
Records an internal error. This informs the operaor that the test was not completed successfully.

Parameters **description** (*str*) – Received data.

Returns None

Return type None

abstract log_fail(*description=''*)
Records a check that failed. This will flag a fuzzing case as a potential bug or anomaly.

Parameters **description** (*str*) – Optional supplementary data.

Returns None

Return type None

abstract log_info(*description*)
Catch-all method for logging test information

Parameters **description** (*str*) – Information.

Returns None

Return type None

abstract log_pass(*description=''*)
Records a check that passed.

Parameters **description** (*str*) – Optional supplementary data..

Returns None

Return type None

abstract log_recv(*data*)
Records data as having been received from the target.

Parameters **data** (*bytes*) – Received data.

Returns None

Return type None

abstract log_send(*data*)
Records data as about to be sent to the target.

Parameters **data** (*bytes*) – Transmitted data

Returns None

Return type None

abstract open_test_case(*test_case_id*, *name*, *index*, **args*, ***kwargs*)
Open a test case - i.e., a fuzzing mutation.

Parameters

- **test_case_id** – Test case name/number. Should be unique.
- **name** (*str*) – Human readable and unique name for test case.
- **index** (*int*) – Numeric index for test case

Returns None

abstract open_test_step(*description*)
Open a test step - e.g., "Fuzzing", "Pre-fuzz", "Response Check."

Parameters **description** – Description of fuzzing step.

Returns None

boofuzz.**IFuzzLoggerBackend**
alias of boofuzz.ifuzz_logger.IFuzzLogger

4.5.2 Text Logging

class boofuzz.**FuzzLoggerText**(*file_handle=<colorama.ansitowin32.StreamWrapper object>*, *bytes_to_str=<function hex_to_hexstr>*)
Bases: boofuzz.ifuzz_logger.IFuzzLogger

This class formats FuzzLogger data for text presentation. It can be configured to output to STDOUT, or to a named file.

Using two FuzzLoggerTexts, a FuzzLogger instance can be configured to output to both console and file.

INDENT_SIZE = 2

close_test()
Called after a test has been completed. Can be used to inform the operator or save the test log.

Param None

Type None

Returns None

Return type None

close_test_case()
Called after a test case has been completed. Can be used to inform the operator or save the test case log.

Param None

Type None

Returns None

Return type None

log_check(*description*)
Records a check on the system under test. AKA "instrumentation check."

Parameters **description** (*str*) – Received data.

Returns None

Return type None

log_error(*description*)
Records an internal error. This informs the operaor that the test was not completed successfully.

Parameters **description** (*str*) – Received data.

Returns None

Return type None

log_fail(*description=''*)
Records a check that failed. This will flag a fuzzing case as a potential bug or anomaly.

Parameters **description** (*str*) – Optional supplementary data.

Returns None

Return type None

log_info(*description*)
Catch-all method for logging test information

Parameters **description** (*str*) – Information.

Returns None

Return type None

log_pass(*description=''*)
Records a check that passed.

Parameters **description** (*str*) – Optional supplementary data..

Returns None

Return type None

log_recv(*data*)
Records data as having been received from the target.

Parameters **data** (*bytes*) – Received data.

Returns None

Return type None

log_send(*data*)
Records data as about to be sent to the target.

Parameters **data** (*bytes*) – Transmitted data

Returns None

Return type None

open_test_case(*test_case_id*, *name*, *index*, **args*, ***kwargs*)
Open a test case - i.e., a fuzzing mutation.

Parameters

- **test_case_id** – Test case name/number. Should be unique.
- **name** (*str*) – Human readable and unique name for test case.
- **index** (*int*) – Numeric index for test case

Returns None

open_test_step(*description*)
Open a test step - e.g., "Fuzzing", "Pre-fuzz", "Response Check."

Parameters **description** – Description of fuzzing step.

Returns None

4.5.3 CSV Logging

class boofuzz.**FuzzLoggerCsv**(*file_handle=<_io.TextIOWrapper name='<stdout>' mode='w' encoding='UTF-8'>*, *bytes_to_str=<function hex_to_hexstr>*)
Bases: boofuzz.ifuzz_logger.IFuzzLogger

This class formats FuzzLogger data for pcap file. It can be configured to output to a named file.

close_test()
Called after a test has been completed. Can be used to inform the operator or save the test log.

Param None

Type None

Returns None

Return type None

close_test_case()
Called after a test case has been completed. Can be used to inform the operator or save the test case log.

Param None

Type None

Returns None

Return type None

log_check(*description*)
Records a check on the system under test. AKA "instrumentation check."

Parameters **description** (*str*) – Received data.

Returns None

Return type None

log_error(*description*)
Records an internal error. This informs the operaor that the test was not completed successfully.

Parameters **description** (*str*) – Received data.

Returns None

Return type None

log_fail(*description=''*)
Records a check that failed. This will flag a fuzzing case as a potential bug or anomaly.

Parameters **description** (*str*) – Optional supplementary data.

Returns None

Return type None

log_info(*description*)
Catch-all method for logging test information

Parameters **description** (*str*) – Information.

Returns None

Return type None

log_pass(*description=''*)
Records a check that passed.

Parameters **description** (*str*) – Optional supplementary data..

Returns None

Return type None

log_recv(*data*)
Records data as having been received from the target.

Parameters **data** (*bytes*) – Received data.

Returns None

Return type None

log_send(*data*)
Records data as about to be sent to the target.

Parameters **data** (*bytes*) – Transmitted data

Returns None

Return type None

open_test_case(*test_case_id*, *name*, *index*, **args*, ***kwargs*)
Open a test case - i.e., a fuzzing mutation.

Parameters

- **test_case_id** – Test case name/number. Should be unique.
- **name** (*str*) – Human readable and unique name for test case.
- **index** (*int*) – Numeric index for test case

Returns None

open_test_step(*description*)
Open a test step - e.g., "Fuzzing", "Pre-fuzz", "Response Check."

Parameters **description** – Description of fuzzing step.

Returns None

4.5.4 Console-GUI Logging

class boofuzz.**FuzzLoggerCurses**(*web_port=26000*, *window_height=40*, *window_width=130*, *auto_scroll=True*, *max_log_lines=500*, *wait_on_quit=True*, *min_refresh_rate=1000*, *bytes_to_str=<function hex_to_hexstr>*)

Bases: boofuzz.ifuzz_logger.IFuzzLogger

This class formats FuzzLogger data for a console GUI using curses. This hasn't been tested on Windows.

INDENT_SIZE = 2

close_test()

Called after a test has been completed. Can be used to inform the operator or save the test log.

Param None

Type None

Returns None

Return type None

close_test_case()

Called after a test case has been completed. Can be used to inform the operator or save the test case log.

Param None

Type None

Returns None

Return type None

log_check(*description*)

Records a check on the system under test. AKA "instrumentation check."

Parameters **description** (*str*) – Received data.

Returns None

Return type None

log_error(*description=''*, *indent_size=2*)

Records an internal error. This informs the operaor that the test was not completed successfully.

Parameters **description** (*str*) – Received data.

Returns None

Return type None

log_fail(*description=''*, *indent_size=2*)

Records a check that failed. This will flag a fuzzing case as a potential bug or anomaly.

Parameters **description** (*str*) – Optional supplementary data.

Returns None

Return type None

log_info(*description*)

Catch-all method for logging test information

Parameters **description** (*str*) – Information.

Returns None

Return type None

log_pass(*description=''*)
Records a check that passed.

Parameters **description** (*str*) – Optional supplementary data..

Returns None

Return type None

log_recv(*data*)
Records data as having been received from the target.

Parameters **data** (*bytes*) – Received data.

Returns None

Return type None

log_send(*data*)
Records data as about to be sent to the target.

Parameters **data** (*bytes*) – Transmitted data

Returns None

Return type None

open_test_case(*test_case_id*, *name*, *index*, **args*, ***kwargs*)
Open a test case - i.e., a fuzzing mutation.

Parameters

- **test_case_id** – Test case name/number. Should be unique.
- **name** (*str*) – Human readable and unique name for test case.
- **index** (*int*) – Numeric index for test case

Returns None

open_test_step(*description*)
Open a test step - e.g., "Fuzzing", "Pre-fuzz", "Response Check."

Parameters **description** – Description of fuzzing step.

Returns None

4.5.5 FuzzLogger Object

class boofuzz.**FuzzLogger**(*fuzz_loggers=None*)
Bases: boofuzz.ifuzz_logger.IFuzzLogger

Takes a list of IFuzzLogger objects and multiplexes logged data to each one.

FuzzLogger also maintains summary failure and error data.

Parameters **fuzz_loggers** (list of *IFuzzLogger*) – IFuzzLogger objects to which to send log data.

close_test()
Called after a test has been completed. Can be used to inform the operator or save the test log.

Param None

Type None

Returns None

Return type None

close_test_case()
Called after a test case has been completed. Can be used to inform the operator or save the test case log.

Param None

Type None

Returns None

Return type None

failure_summary()
Return test summary string based on fuzz logger results.

Returns Test summary string, may be multi-line.

log_check(*description*)
Records a check on the system under test. AKA "instrumentation check."

Parameters **description** (*str*) – Received data.

Returns None

Return type None

log_error(*description*)
Records an internal error. This informs the operaor that the test was not completed successfully.

Parameters **description** (*str*) – Received data.

Returns None

Return type None

log_fail(*description=''*)
Records a check that failed. This will flag a fuzzing case as a potential bug or anomaly.

Parameters **description** (*str*) – Optional supplementary data.

Returns None

Return type None

log_info(*description*)
Catch-all method for logging test information

Parameters **description** (*str*) – Information.

Returns None

Return type None

log_pass(*description=''*)
Records a check that passed.

Parameters **description** (*str*) – Optional supplementary data..

Returns None

Return type None

log_recv(*data*)
Records data as having been received from the target.

Parameters **data** (*bytes*) – Received data.

Returns None

Return type None

log_send(*data*)
Records data as about to be sent to the target.

Parameters **data** (*bytes*) – Transmitted data

Returns None

Return type None

open_test_case(*test_case_id*, *name*, *index*, **args*, ***kwargs*)
Open a test case - i.e., a fuzzing mutation.

Parameters

- **test_case_id** – Test case name/number. Should be unique.
- **name** (*str*) – Human readable and unique name for test case.
- **index** (*int*) – Numeric index for test case

Returns None

open_test_step(*description*)
Open a test step - e.g., "Fuzzing", "Pre-fuzz", "Response Check."

Parameters **description** – Description of fuzzing step.

Returns None

4.6 Static Protocol Definition

Static functions are used in boofuzz to assemble messages for a protocol definition. They may be obsoleted in future releases by a less static approach to message construction. For now, you can see the *Quickstart* guide for an intro.

Requests are messages, Blocks are chunks within a message, and Primitives are the elements (bytes, strings, numbers, checksums, etc.) that make up a Block/Request.

4.6.1 Request Manipulation

boofuzz.**s_initialize**(*name*)
Initialize a new block request. All blocks / primitives generated after this call apply to the named request. Use s_switch() to jump between factories.

Parameters **name** (*str*) – Name of request

boofuzz.**s_get**(*name=None*)
Return the request with the specified name or the current request if name is not specified. Use this to switch from global function style request manipulation to direct object manipulation. Example:

```
req = s_get("HTTP BASIC")
print(req.num_mutations())
```

The selected request is also set as the default current. (ie: s_switch(name) is implied).

Parameters **name** (*str*) – (Optional, def=None) Name of request to return or current request if name is None.

Return type blocks.Request

Returns The requested request.

boofuzz.**s_mutate**()
Mutate the current request and return False if mutations are exhausted, in which case the request has been reverted back to its normal form.

Return type bool

Returns True on mutation success, False if mutations exhausted.

boofuzz.**s_num_mutations**()
Determine the number of repetitions we will be making.

Return type int

Returns Number of mutated forms this primitive can take.

boofuzz.**s_render**()
Render out and return the entire contents of the current request.

Return type Raw

Returns Rendered contents

boofuzz.**s_switch**(*name*)
Change the current request to the one specified by "name".

Parameters **name** (*str*) – Name of request

4.6.2 Block Manipulation

boofuzz.**s_block**(*name*, *group=None*, *encoder=None*, *dep=None*, *dep_value=None*, *dep_values=()*, *dep_compare='=='*)
Open a new block under the current request. The returned instance supports the "with" interface so it will be automatically closed for you:

```
with s_block("header"):
    s_static("\x00\x01")
    if s_block_start("body"):
        ...
```

Parameters

- **name** (*str*) – Name of block being opened
- **group** (*str*) – (Optional, def=None) Name of group to associate this block with
- **encoder** (*Function Pointer*) – (Optional, def=None) Optional pointer to a function to pass rendered data to prior to return
- **dep** (*str*) – (Optional, def=None) Optional primitive whose specific value this block is dependant on
- **dep_value** (*Mixed*) – (Optional, def=None) Value that field "dep" must contain for block to be rendered
- **dep_values** (*List of Mixed Types*) – (Optional, def=[]) Values that field "dep" may contain for block to be rendered
- **dep_compare** (*str*) – (Optional, def="==") Comparison method to use on dependency (==, !=, >, >=, <, <=)

boofuzz.**s_block_start**(*name*, **args*, ***kwargs*)

Open a new block under the current request. This routine always returns an instance so you can make your fuzzer pretty with indenting:

```
if s_block_start("header"):
    s_static("\x00\x01")
    if s_block_start("body"):
        ...
s_block_close()
```

:note Prefer using s_block to this function directly :see s_block

boofuzz.**s_block_end**(*name=None*)

Close the last opened block. Optionally specify the name of the block being closed (purely for aesthetic purposes).

Parameters **name** (*str*) – (Optional, def=None) Name of block to closed.

boofuzz.**s_checksum**(*block_name*, *algorithm='crc32'*, *length=0*, *endian='<'*, *fuzzable=True*, *name=None*, *ipv4_src_block_name=None*, *ipv4_dst_block_name=None*)

Create a checksum block bound to the block with the specified name. You *can not* create a checksum for any currently open blocks.

Parameters

- **block_name** (*str*) – Name of block for checksum calculations
- **algorithm** (*str, function*) – (Optional, def=crc32) Checksum algorithm to use. (crc32, crc32c, adler32, md5, sha1, ipv4, udp) Pass a function to use a custom algorithm. This function has to take and return byte-type data.
- **length** (*int*) – (Optional, def=0) Length of checksum, auto-calculated by default. Must be specified manually when using a custom algorithm.
- **endian** (*Character*) – (Optional, def=LITTLE_ENDIAN) Endianness of the bit field (LITTLE_ENDIAN: <, BIG_ENDIAN: >)
- **fuzzable** (*bool*) – (Optional, def=True) Enable/disable fuzzing.
- **name** (*str*) – Name of this checksum field
- **ipv4_src_block_name** (*str*) – Required for 'udp' algorithm. Name of block yielding IPv4 source address.
- **ipv4_dst_block_name** (*str*) – Required for 'udp' algorithm. Name of block yielding IPv4 destination address.

boofuzz.**s_repeat**(*block_name*, *min_reps=0*, *max_reps=None*, *step=1*, *variable=None*, *fuzzable=True*, *name=None*)

Repeat the rendered contents of the specified block cycling from min_reps to max_reps counting by step. By default renders to nothing. This block modifier is useful for fuzzing overflows in table entries. This block modifier MUST come after the block it is being applied to.

See Aliases: s_repeater()

Parameters

- **block_name** (*str*) – Name of block to repeat
- **min_reps** (*int*) – (Optional, def=0) Minimum number of block repetitions
- **max_reps** (*int*) – (Optional, def=None) Maximum number of block repetitions
- **step** (*int*) – (Optional, def=1) Step count between min and max reps

- **variable** (*Sulley Integer Primitive*) – (Optional, def=None) An integer primitive which will specify the number of repitions
- **fuzzable** (*bool*) – (Optional, def=True) Enable/disable fuzzing of this primitive
- **name** (*str*) – (Optional, def=None) Specifying a name gives you direct access to a primitive

boofuzz.**s_size**(*block_name*, *offset=0*, *length=4*, *endian='<'*, *output_format='binary'*, *inclusive=False*, *signed=False*, *math=None*, *fuzzable=True*, *name=None*)

Create a sizer block bound to the block with the specified name. You *can not* create a sizer for any currently open blocks.

See Aliases: s_sizer()

Parameters

- **block_name** (*str*) – Name of block to apply sizer to
- **offset** (*int*) – (Optional, def=0) Offset to calculated size of block
- **length** (*int*) – (Optional, def=4) Length of sizer
- **endian** (*Character*) – (Optional, def=LITTLE_ENDIAN) Endianess of the bit field (LITTLE_ENDIAN: <, BIG_ENDIAN: >)
- **output_format** (*str*) – (Optional, def=binary) Output format, "binary" or "ascii"
- **inclusive** (*bool*) – (Optional, def=False) Should the sizer count its own length?
- **signed** (*bool*) – (Optional, def=False) Make size signed vs. unsigned (applicable only with format="ascii")
- **math** (*Function*) – (Optional, def=None) Apply the mathematical operations defined in this function to the size
- **fuzzable** (*bool*) – (Optional, def=True) Enable/disable fuzzing of this sizer
- **name** (*str*) – Name of this sizer field

boofuzz.**s_update**(*name*, *value*)

Update the value of the named primitive in the currently open request.

Parameters

- **name** (*str*) – Name of object whose value we wish to update
- **value** (*Mixed*) – Updated value

4.6.3 Primitive Definition

boofuzz.**s_binary**(*value*, *name=None*)

Parse a variable format binary string into a static value and push it onto the current block stack.

Parameters

- **value** (*str*) – Variable format binary string
- **name** (*str*) – (Optional, def=None) Specifying a name gives you direct access to a primitive

boofuzz.**s_delim**(*value*, *fuzzable=True*, *name=None*)

Push a delimiter onto the current block stack.

Parameters

- **value** (*Character*) – Original value
- **fuzzable** (*bool*) – (Optional, def=True) Enable/disable fuzzing of this primitive
- **name** (*str*) – (Optional, def=None) Specifying a name gives you direct access to a primitive

boofuzz.**s_group**(*name*, *values*, *default_value=None*)

This primitive represents a list of static values, stepping through each one on mutation. You can tie a block to a group primitive to specify that the block should cycle through all possible mutations for *each* value within the group. The group primitive is useful for example for representing a list of valid opcodes.

Parameters

- **name** (*str*) – Name of group
- **values** (*List or raw data*) – List of possible raw values this group can take.
- **default_value** – (Optional, def=None) Specifying a value when fuzzing() is complete

boofuzz.**s_lego**(*lego_type*, *value=None*, *options=()*)

Legos are pre-built blocks... TODO: finish this doc

Parameters

- **lego_type** (*str*) – Function that represents a lego
- **value** – Original value
- **options** – Options to pass to lego.

boofuzz.**s_random**(*value*, *min_length*, *max_length*, *num_mutations=25*, *fuzzable=True*, *step=None*, *name=None*)

Generate a random chunk of data while maintaining a copy of the original. A random length range can be specified. For a static length, set min/max length to be the same.

Parameters

- **value** (*Raw*) – Original value
- **min_length** (*int*) – Minimum length of random block
- **max_length** (*int*) – Maximum length of random block
- **num_mutations** (*int*) – (Optional, def=25) Number of mutations to make before reverting to default
- **fuzzable** (*bool*) – (Optional, def=True) Enable/disable fuzzing of this primitive
- **step** (*int*) – (Optional, def=None) If not null, step count between min and max reps, otherwise random
- **name** (*str*) – (Optional, def=None) Specifying a name gives you direct access to a primitive

boofuzz.**s_static**(*value*, *name=None*)

Push a static value onto the current block stack.

See Aliases: s_dunno(), s_raw(), s_unknown()

Parameters

- **value** (*Raw*) – Raw static data
- **name** (*str*) – (Optional, def=None) Specifying a name gives you direct access to a primitive

boofuzz.**s_string**(*value*, *size=-1*, *padding=b'\x00'*, *encoding='ascii'*, *fuzzable=True*, *max_len=0*, *name=None*)

Push a string onto the current block stack.

Parameters

- **value** (*str*) – Default string value
- **size** (*int*) – (Optional, def=-1) Static size of this field, leave -1 for dynamic.
- **padding** (*Character*) – (Optional, def="x00") Value to use as padding to fill static field size.
- **encoding** (*str*) – (Optonal, def="ascii") String encoding, ex: utf_16_le for Microsoft Unicode.
- **fuzzable** (*bool*) – (Optional, def=True) Enable/disable fuzzing of this primitive
- **max_len** (*int*) – (Optional, def=0) Maximum string length
- **name** (*str*) – (Optional, def=None) Specifying a name gives you direct access to a primitive

boofuzz.**s_from_file**(*value*, *encoding='ascii'*, *fuzzable=True*, *max_len=0*, *name=None*, *filename=None*)

Push a value from file onto the current block stack.

Parameters

- **value** (*str*) – Default string value
- **encoding** (*str*) – (DEPRECIATED, def="ascii") String encoding, ex: utf_16_le for Microsoft Unicode.
- **fuzzable** (*bool*) – (Optional, def=True) Enable/disable fuzzing of this primitive
- **max_len** (*int*) – (Optional, def=0) Maximum string length
- **name** (*str*) – (Optional, def=None) Specifying a name gives you direct access to a primitive
- **filename** (*str*) – (Mandatory) Specify filename where to read fuzz list

boofuzz.**s_bit_field**(*value*, *width*, *endian='<'*, *output_format='binary'*, *signed=False*, *full_range=False*, *fuzzable=True*, *name=None*)

Push a variable length bit field onto the current block stack.

See Aliases: s_bit(), s_bits()

Parameters

- **value** (*int*) – Default integer value
- **width** (*int*) – Width of bit fields
- **endian** (*Character*) – (Optional, def=LITTLE_ENDIAN) Endianess of the bit field (LITTLE_ENDIAN: <, BIG_ENDIAN: >)
- **output_format** (*str*) – (Optional, def=binary) Output format, "binary" or "ascii"
- **signed** (*bool*) – (Optional, def=False) Make size signed vs. unsigned (applicable only with format="ascii")
- **full_range** (*bool*) – (Optional, def=False) If enabled the field mutates through *all* possible values.
- **fuzzable** (*bool*) – (Optional, def=True) Enable/disable fuzzing of this primitive

- **name** (*str*) – (Optional, def=None) Specifying a name gives you direct access to a primitive

boofuzz.**s_byte**(*value*, *endian='<'*, *output_format='binary'*, *signed=False*, *full_range=False*, *fuzzable=True*, *name=None*)

Push a byte onto the current block stack.

See Aliases: s_char()

Parameters

- **value** (*int|str*) – Default integer value
- **endian** (*Character*) – (Optional, def=LITTLE_ENDIAN) Endianess of the bit field (LITTLE_ENDIAN: <, BIG_ENDIAN: >)
- **output_format** (*str*) – (Optional, def=binary) Output format, "binary" or "ascii"
- **signed** (*bool*) – (Optional, def=False) Make size signed vs. unsigned (applicable only with format="ascii")
- **full_range** (*bool*) – (Optional, def=False) If enabled the field mutates through *all* possible values.
- **fuzzable** (*bool*) – (Optional, def=True) Enable/disable fuzzing of this primitive
- **name** (*str*) – (Optional, def=None) Specifying a name gives you direct access to a primitive

boofuzz.**s_bytes**(*value*, *size=None*, *padding=b'\x00'*, *fuzzable=True*, *max_len=None*, *name=None*)

Push a bytes field of arbitrary length onto the current block stack.

Parameters

- **value** (*bytes*) – Default binary value
- **size** (*int*) – (Optional, def=None) Static size of this field, leave None for dynamic.
- **padding** (*chr*) – (Optional, def=b"x00") Value to use as padding to fill static field size.
- **fuzzable** (*bool*) – (Optional, def=True) Enable/disable fuzzing of this primitive
- **max_len** (*int*) – (Optional, def=None) Maximum string length
- **name** (*str*) – (Optional, def=None) Specifying a name gives you direct access to a primitive

boofuzz.**s_word**(*value*, *endian='<'*, *output_format='binary'*, *signed=False*, *full_range=False*, *fuzzable=True*, *name=None*)

Push a word onto the current block stack.

See Aliases: s_short()

Parameters

- **value** (*int*) – Default integer value
- **endian** (*chr*) – (Optional, def=LITTLE_ENDIAN) Endianess of the bit field (LITTLE_ENDIAN: <, BIG_ENDIAN: >)
- **output_format** (*str*) – (Optional, def=binary) Output format, "binary" or "ascii"
- **signed** (*bool*) – (Optional, def=False) Make size signed vs. unsigned (applicable only with format="ascii")
- **full_range** (*bool*) – (Optional, def=False) If enabled the field mutates through *all* possible values.

- **fuzzable** (*bool*) – (Optional, def=True) Enable/disable fuzzing of this primitive
- **name** (*str*) – (Optional, def=None) Specifying a name gives you direct access to a primitive

boofuzz.**s_dword**(*value*, *endian='<'*, *output_format='binary'*, *signed=False*, *full_range=False*, *fuzzable=True*, *name=None*)

Push a double word onto the current block stack.

See Aliases: s_long(), s_int()

Parameters

- **value** (*int*) – Default integer value
- **endian** (*Character*) – (Optional, def=LITTLE_ENDIAN) Endianess of the bit field (LITTLE_ENDIAN: <, BIG_ENDIAN: >)
- **output_format** (*str*) – (Optional, def=binary) Output format, "binary" or "ascii"
- **signed** (*bool*) – (Optional, def=False) Make size signed vs. unsigned (applicable only with format="ascii")
- **full_range** (*bool*) – (Optional, def=False) If enabled the field mutates through *all* possible values.
- **fuzzable** (*bool*) – (Optional, def=True) Enable/disable fuzzing of this primitive
- **name** (*str*) – (Optional, def=None) Specifying a name gives you direct access to a primitive

boofuzz.**s_qword**(*value*, *endian='<'*, *output_format='binary'*, *signed=False*, *full_range=False*, *fuzzable=True*, *name=None*)

Push a quad word onto the current block stack.

See Aliases: s_double()

Parameters

- **value** (*int*) – Default integer value
- **endian** (*Character*) – (Optional, def=LITTLE_ENDIAN) Endianess of the bit field (LITTLE_ENDIAN: <, BIG_ENDIAN: >)
- **output_format** (*str*) – (Optional, def=binary) Output format, "binary" or "ascii"
- **signed** (*bool*) – (Optional, def=False) Make size signed vs. unsigned (applicable only with format="ascii")
- **full_range** (*bool*) – (Optional, def=False) If enabled the field mutates through *all* possible values.
- **fuzzable** (*bool*) – (Optional, def=True) Enable/disable fuzzing of this primitive
- **name** (*str*) – (Optional, def=None) Specifying a name gives you direct access to a primitive

4.6.4 Making Your Own Block/Primitive

Now I know what you're thinking: "With that many sweet primitives and blocks available, what else could I ever conceivably need? And yet, I am urged by joy to contribute my own sweet blocks!"

To make your own block/primitive:

1. Create an object that implements `IFuzzable`.
2. Create an accompanying static primitive function. See boofuzz's *__init__.py* file for examples.
3. ???
4. Profit!

If your block depends on references to other blocks, the way a checksum or length field depends on other parts of the message, see the `Size` source code for an example of how to avoid recursion issues. Or otherwise be careful. :)

4.7 Other Modules

4.7.1 EventHook

class boofuzz.event_hook.**EventHook**
Bases: `object`

An EventHook that registers events using +=and -=.

Based on spassig's solution here: http://stackoverflow.com/a/1094423/461834

fire(**args*, ***kwargs*)
Call each event handler in sequence.

@param args: Forwarded to event handler. @param kwargs: Forwarded to event handler.

@return: None

4.7.2 Helpers

boofuzz.helpers.**calculate_four_byte_padding**(*string*, *character='\x00'*)

boofuzz.helpers.**crc16**(*string*, *value=0*)
CRC-16 poly: p(x) = x**16 + x**15 + x**2 + 1

@param string: Data over which to calculate crc. @param value: Initial CRC value.

boofuzz.helpers.**crc32**(*string*)

boofuzz.helpers.**format_log_msg**(*msg_type*, *description=None*, *data=None*, *indent_size=2*, *timestamp=None*, *truncated=False*, *format_type='terminal'*)

boofuzz.helpers.**format_msg**(*msg*, *indent_level*, *indent_size*, *timestamp=None*)

boofuzz.helpers.**get_boofuzz_version**(*boofuzz_class*)
Parses __init__.py for a version string and returns it like 'v0.0.0'

Parameters **boofuzz_class** (*class*) – Any boofuzz class in the same dir as the __init__ class.

Return type str

Returns Boofuzz version as string

boofuzz.helpers.**get_max_udp_size**()
Crazy CTypes magic to do a getsockopt() which determines the max UDP payload size in a platform-agnostic way.

Deprecated since version 0.2.0: Use *UDPSocketConnection.max_payload()* instead.

Returns The maximum length of a UDP packet the current platform supports

Return type int

boofuzz.helpers.**get_time_stamp**()

boofuzz.helpers.**hex_str**(*s*)
Returns a hex-formatted string based on s.

Parameters **s** (*bytes*) – Some string.

Returns Hex-formatted string representing s.

Return type str

boofuzz.helpers.**hex_to_hexstr**(*input_bytes*)
Render input_bytes as ASCII-encoded hex bytes, followed by a best effort utf-8 rendering.

Parameters **input_bytes** (*bytes*) – Arbitrary bytes

Returns Printable string

Return type str

boofuzz.helpers.**ip_str_to_bytes**(*ip*)
Convert an IP string to a four-byte bytes.

Parameters **ip** – IP address string, e.g. '127.0.0.1'

:return 4-byte representation of ip, e.g. b'' :rtype bytes

:raises ValueError if ip is not a legal IP address.

boofuzz.helpers.**ipv4_checksum**(*msg*)
Return IPv4 checksum of msg. :param msg: Message to compute checksum over. :type msg: bytes

Returns IPv4 checksum of msg.

Return type int

boofuzz.helpers.**mkdir_safe**(*directory_name*)

boofuzz.helpers.**pause_for_signal**()
Pauses the current thread in a way that can still receive signals like SIGINT from Ctrl+C.

Implementation notes:

- Linux uses signal.pause()
- Windows uses a loop that sleeps for 1 ms at a time, allowing signals to interrupt the thread fairly quickly.

Returns None

Return type None

boofuzz.helpers.**str_to_bytes**(*value*)

boofuzz.helpers.**udp_checksum**(*msg*, *src_addr*, *dst_addr*)
Return UDP checksum of msg.

Recall that the UDP checksum involves creating a sort of pseudo IP header. This header requires the source and destination IP addresses, which this function takes as parameters.

If msg is too big, the checksum is undefined, and this method will truncate it for the sake of checksum calculation. Note that this means the checksum will be invalid. This loosey goosey error checking is done to support fuzz tests which at times generate huge, invalid packets.

Parameters

- **msg** (*bytes*) – Message to compute checksum over.
- **src_addr** (*bytes*) – Source IP address – 4 bytes.
- **dst_addr** (*bytes*) – Destination IP address – 4 bytes.

Returns UDP checksum of msg.

Return type int

boofuzz.helpers.**uuid_bin_to_str**(*uuid*)
Convert a binary UUID to human readable string.

@param uuid: bytes representing UUID.

boofuzz.helpers.**uuid_str_to_bin**(*uuid*)
Converts a UUID string to binary form.

Expected string input format is same as uuid_bin_to_str()'s output format.

Ripped from Core Impacket.

Parameters **uuid** (*str*) – UUID string to convert to bytes.

Returns UUID as bytes.

Return type bytes

4.7.3 IP Constants

This file contains constants for the IPv4 protocol.

Changed in version 0.2.0: ip_constants has been moved into the connections subpackage. The full path is now boofuzz.connections.ip_constants

boofuzz.connections.ip_constants.**UDP_MAX_LENGTH_THEORETICAL = 65535**
Theoretical maximum length of a UDP packet, based on constraints in the UDP packet format. WARNING! a UDP packet cannot actually be this long in the context of IPv4!

boofuzz.connections.ip_constants.**UDP_MAX_PAYLOAD_IPV4_THEORETICAL = 65507**
Theoretical maximum length of a UDP payload based on constraints in the UDP and IPv4 packet formats. WARNING! Some systems may set a payload limit smaller than this.

4.7.4 PED-RPC

Boofuzz provides an RPC primitive to host monitors on remote machines. The main boofuzz instance acts as a client that connects to (remotely) running RPC server instances, transparently calling functions that are called on the instance of the client on the server instance and returning their result as a python object. As a general rule, data that's passed over the RPC interface needs to be able to be pickled.

Note that PED-RPC provides no authentication or authorization in any form. It is advisable to only run it on trusted networks.

class boofuzz.monitors.pedrpc.**Client**(*host*, *port*)
Bases: object

on_new_server(*new_server*)
Override this Method in a child class to be notified when the RPC server was restarted.

class boofuzz.monitors.pedrpc.**Server**(*host*, *port*)
Bases: object

The main PED-RPC Server class. To implement an RPC server, inherit from this class. Call serve_forever to start listening for RPC commands.

serve_forever()

stop()

4.7.5 DCE-RPC

boofuzz.utils.dcerpc.**bind**(*uuid*, *version*)
Generate the data necessary to bind to the specified interface.

boofuzz.utils.dcerpc.**bind_ack**(*data*)
Ensure the data is a bind ack and that the

boofuzz.utils.dcerpc.**request**(*opnum*, *data*)
Return a list of packets broken into 5k fragmented chunks necessary to make the RPC request.

4.7.6 Crash binning

@author: Pedram Amini @license: GNU General Public License 2.0 or later @contact: pedram.amini@gmail.com @organization: www.openrce.org

class boofuzz.utils.crash_binning.**CrashBinStruct**
Bases: object

class boofuzz.utils.crash_binning.**CrashBinning**
Bases: object

@todo: Add MySQL import/export.

bins = {}

crash_synopsis(*crash=None*)
For the supplied crash, generate and return a report containing the disassemly around the violating address, the ID of the offending thread, the call stack and the SEH unwind. If not crash is specified, then call through to last_crash_synopsis() which returns the same information for the last recorded crash.

@see: crash_synopsis()

@type crash: CrashBinStruct @param crash: (Optional, def=None) Crash object to generate report on

@rtype: str @return: Crash report

export_file(*file_name*)
Dump the entire object structure to disk.

@see: import_file()

@type file_name: str @param file_name: File name to export to

@rtype: CrashBinning @return: self

import_file(*file_name*)
Load the entire object structure from disk.

@see: export_file()

@type file_name: str @param file_name: File name to import from

@rtype: CrashBinning @return: self

last_crash = None

last_crash_synopsis()
For the last recorded crash, generate and return a report containing the disassemly around the violating address, the ID of the offending thread, the call stack and the SEH unwind.

@see: crash_synopsis()

@rtype: String @return: Crash report

pydbg = None

record_crash(*pydbg*, *extra=None*)
Given a PyDbg instantiation that at the current time is assumed to have "crashed" (access violation for example) record various details such as the disassemly around the violating address, the ID of the offending thread, the call stack and the SEH unwind. Store the recorded data in an internal dictionary, binning them by the exception address.

@type pydbg: pydbg @param pydbg: Instance of pydbg @type extra: Mixed @param extra: (Optional, Def=None) Whatever extra data you want to store with this bin

4.8 Changelog

4.8.1 Upcoming

Features

Fixes

- Fixed UDPSocketConnection data truncation when sending more data than the socket supports.
- Fixed execution of procmon stop_commands.
- Fixed TCP and SSL server connections.

4.8.2 v0.2.0

Features

- Rewrote and split the SocketConnection class into individual classes per socket type.
- *SocketConnection* is now deprecated. Use the classes derived from *BaseSocketConnection* instead.
- Added support for receiving on raw Layer 2 and Layer 3 connections.
- Layer 2 and Layer 3 connections may now use arbitrary payload / MTU sizes.
- Moved connection related modules into new *connections* submodule.
- Added the ability to repeat sending of packages within a given time or count.
- Added optional timeout and threshold to quit infinite connection retries.
- Reworked Monitors, consolidated interface. Breaking change: session no longer has netmon_options and procmon_options.
- *SessionInfo* has had attributes renamed; procmon_results and netmon_results are deprecated and now aliases for monitor_results and monitor_data respectively.
- New *BoofuzzFailure* exception type allows callback methods to signal a failure that should halt the current test case.
- Added *capture_output* option to process monitor to capture target process stderr/stdout .
- Added post-start-target callbacks (called every time a target is started or restarted).
- Added method to gracefully stop PED-RPC Server.
- Added new boofuzz logo and favicon to docs and webinterface.
- Added *FileConnection* to dump messages to files.
- Removed deprecated session arguments *fuzz_data_logger*, *log_level*, *logfile*, *logfile_level* and *log()*.
- Removed deprecated logger *FuzzLoggerFile*.
- *crc32c* is no longer a required package. Install manually if needed.

Fixes

- Fixed size of s_size block when output is ascii.
- Fixed issue with tornado on Python 3.8 and Windows.
- Fixed various potential type errors.
- Renamed *requests* folder to *request_definitions* because it shadowed the name of the *requests* python module.
- Examples are up to date with current Boofuzz version.
- Modified timings on serial_connection unit tests to improve test reliability.
- Refactored old unit-tests.
- Fixed network monitor compatibility with Python 3.
- Minor console GUI optimizations.
- Fixed crash_threshold_element handling if blocks are used.
- Fixed many bugs in which a failure would not stop the test case evaluation.

4.8.3 v0.1.6

Features

- New primitive *s_bytes* which fuzzes an arbitrary length binary value (similiar to *s_string*).
- We are now using *Black* for code style standardization.
- Compatibility for Python 3.8
- Added crc32c as checksum algorithm (Castagnoli).
- Added favicon for web interface.
- Pushed Tornado to 5.x and unpinned Flask.

Fixes

- Test cases were not being properly closed when using the check_message() functionality.
- Some code style changes to meet PEP8.
- *s_group* primitive was not accepting empty default value.
- Timeout during opening TCP connection now raises BoofuzzTargetConnectionFailedError exception.
- SSL/TLS works again. See *examples/fuzz-ssl-server.py* and *examples/fuzz-ssl-client.py*.
- Dropped six.binary_type in favor of b"" format.
- Fixed process monitor handling of backslashes in Windows start commands.
- Fixed and documented *boo open*.
- Fixed receive function in *fuzz_logger_curses*.
- Installing boofuzz with *sudo* is no longer recommended, use the *–user* option of pip instead.
- Fixed setting socket timeout options on Windows.
- If all sockets are exhausted, repeatedly try fuzzing for 4 minutes before failing.
- Fixed CSV logger send and receive data decoding.
- Handle SSL-related exception. Added *ignore_connection_ssl_errors* session attribute that can be set to True to ignore SSL-related error on a test case.
- Fixed *s_from_file* decoding in Python 2 (the encoding parameter is now depreciated).
- Updated documentation of *s_checksum*. It is possible to use a custom algorithm with this block.

4.8.4 v0.1.5

Features

- New curses logger class to provide a console gui similar to the webinterface. Use the session option *console_gui* to enable it. This has not been tested under Windows!
- Compatibility for Python 3
- Large test cases are now truncated, unless a failure is detected.
- When a target fails to respond after restart, boofuzz will now continue to restart instead of crashing.

- New Session option *keep_web_open* to allow analyzing the test results after test completion.
- Process monitor creates new crash file for each run by default.
- Long lines now wrap in web view; longer lines no longer need to be truncated.
- Process monitor now stores crash bins in JSON format instead of pickled format.
- Process monitor in Windows will use *taskkill -F* if *taskkill* fails.

Fixes

- Web server no longer crashes when asked for a non-existing test case.
- EINPROGRESS socket error is now handled while opening a socket (note: this sometimes-transient error motivated the move to retry upon connection failure)

4.8.5 v0.1.4

Features

- New Session options *restart_callbacks*, *pre_send_callbacks*, and *post_test_case_callbacks* to hand over custom callback functions.
- New Session option *fuzz_db_keep_only_n_pass_cases*. This allowes saving only n test cases preceding a failure or error to the database.
- Added logic to find next available port for web interface or disable the web interface.
- Removed sleep logs when sleep time is zero.
- Added option to reuse the connection to the target.

Fixes

- Windows process monitor now handles combination of proc_name and/or start_commands more reasonably
- Windows process monitor handles certain errors more gracefully
- Fixed target close behavior so post send callbacks can use the target.
- Fixed a dependency issue in installation.

4.8.6 v0.1.3

Features

- Socket Connections now allow client fuzzing.
- Log only the data actually sent, when sending is truncated. Helps reduce database size, especially when fuzzing layer 2 or 3.
- *Target recv* function now accepts a *max_recv_bytes* argument.

Fixes

- Fixed install package – now includes JavaScript files.

4.8.7 v0.1.2

Features

- Clearer error message when procmon is unavailable at fuzz start.
- Web UI now refreshes current case even when snap-to-current-test-case is disabled.

Fixes

- Web UI no longer permits negative test cases.
- Fix Windows procmon regression.
- Minor fixes and UI tweaks.

4.8.8 v0.1.1

Features

- New *boo open* command can open and inspect saved database log files.
- Unix procmon now saves coredumps by default.
- Improved "Cannot connect to target" error message.
- Improved API for registering callbacks.
- Made the global *REQUESTS* map available in top level boofuzz package.

Fixes

- Handle exceptions when opening crash bin files in process monitor.
- Fix Block.__len__ to account for custom encoder.

4.8.9 v0.1.0

Features

- **Web UI**
  - Statistics now auto-update.
  - Test case logs now stream on the main page.
  - Cool left & right arrow buttons to move through test case
- New `Session` parameter `receive_data_after_fuzz`. Controls whether to execute a receive step after sending fuzz messages. Defaults to False. This significantly speeds up tests in which the target tends not to respond to invalid messages.

Fixes

- Text log output would include double titles, e.g. "Test Step: Test Step: ..."

4.8.10 v0.0.13

Features

- **Web UI**
  - Test case numbers are now clickable and link to test case detail view.
  - Test case details now in color!
- **`FuzzLoggerDB`**
  - Added FuzzLoggerDB to allow querying of test results during and after test run. Saves results in a SQLite file.
  - Added `Session.open_test_run()` to read test results database from previous test run.
- New `Session.feature_check()` method to verify protocol functionality before fuzzing.
- **Process Monitor**
  - Unify process monitor command line interface between Unix and Windows.
  - Added procmon option `proc_name` to support asynchronously started target processes.
  - procmon is now checked for errors before user `post_send()` is called, reducing redundant error messages.
  - Improved procmon logging.
  - Process monitor gives more helpful error messages when running 64-bit application (unsupported) or when a process is killed before being attached
- **Logging Improvements**
  - `Target` `open()` and `close()` operations are now logged.
  - Added some optional debug output from boofuzz runtime.
  - Improve capability and logging of messages' `callback` methods.
- **New `Session` & Connection Options**
  - Add `Session` `receive_data_after_each_request` option to enable disabling of data receipt after messages are sent.
  - `Session` `skip` argument replaced with `index_start` and `index_end`.
  - `Session` now has separate crash thresholds for elements/blocks and nodes/messages.
  - Give `SocketConnection` separate timeouts for `send()`/`recv()`.
- **Ease of Use**
  - `Target.recv()` now has a default `max_bytes` value.
  - Added `DEFAULT_PROCMON_PORT` constant.
  - `Session.post_send()`'s `sock` parameter now deprecated (use `target` instead).

Fixes

- Fixed bug in which failures were not recognized.
- `BitField` blocks with ASCII format reported incorrect sizes.
- Fixed bug in `s_update`.
- Handle socket errors that were getting missed.
- Fixed process monitor logging when providing more or less than 1 stop/start commands.
- Show graceful error on web requests for non-existent test cases.
- `get_max_udp_size()` was crashing in Windows.
- `String` padding was not always being applied.
- `String` was not accepting unicode strings in `value` parameter.
- `String` was skipping valid mutations and reporting wrong `num_mutations()` when `size` parameter was used.
- Unix and Windows process monitors now share much more code.

Development

- Added unit tests for `BitField`.
- Cleaned up CSS on web pages.
- Added a unit test to verify restart on failure behavior

4.8.11 0.0.12

Features

- Test cases now have descriptive names
- Added Session methods to fuzz a test cae by name: `fuzz_by_name` and `fuzz_single_node_by_path`

Fixes

- Fixed test case numbers when using `fuzz_single_case`

4.8.12 0.0.11

Features

- Set Session `check_data_received_each_request` to False to disable receive after send.

Fixes

- Dosctring format fixes.

4.8.13 0.0.10

Features

- Add Session ignore_connection_reset parameter to suppress ECONNRESET errors.
- Add Session ignore_connection_aborted parameter to suppress ECONNABORTED errors.

Fixes

- Fix Session class docstring formats.

4.8.14 0.0.9

Features

- `s_size` is now fuzzable by default.
- Add new s_fuzz_list primitive to read fuzz value from files.
- Add new FuzzLoggerCsv to write log in CSV format

Fixes

- Fixed: Add missing dummy value for custom checksum, allowing recursive uses of length/checksum (issue #107)

4.8.15 0.0.8

Features

- Console output - now with colors!
- process_monitor_unix.py: added option to move coredumps for later analysis.
- The process monitor (procmon) now tracks processes by PID by default rather than searching by name. Therefore, stop_commands and proc_name are no longer required.
- SIGINT (AKA Ctrl+C) now works to close both boofuzz and process_monitor.py (usually).
- Made Unix procmon more compatible with Windows.
- Improved procmon debugger error handling, e.g., when running 64-bit apps.
- Windows procmon now runs even if pydbg fails.
- Added `--help` parameter to process monitor.
- Target class now takes `procmon` and `procmon_options` in constructor.
- Added example fuzz scripts.

Fixes

- SIGINT (AKA Ctrl+C) now works to close both boofuzz and process_monitor.py (usually).
- Fixed: The pedrpc module was not being properly included in imports.
- Made process_monitor.py `--crash_bin` optional (as documented).
- Improved procmon behavior when certain parameters aren't given.
- Improved procmon error handling.
- Fixed a bug in which the procmon would not properly restart a target that had failed without crashing.

4.8.16 0.0.7

Features

- Added several command injection strings from fuzzdb.
- Blocks can now be created and nested using `with s_block("my-block"):`

Fixes

- Fixed pydot import error message

4.8.17 0.0.6

Features

- Added `Request.original_value()` function to render the request as if it were not fuzzed. This will help enable reuse of a fuzz definition to generate valid requests.
- `SocketConnection` can now send and receive UDP broadcast packets using the `udp_broadcast` constructor parameter.
- `Target.recv()` now logs an entry before receiving data, in order to help debug receiving issues.

Fixes

- Maximum UDP payload value was incorrect, causing crashes for tests running over UDP. It now works on some systems, but the maximum value may be too high for systems that set it lower than the maximum possible value, 65507.
- `SocketConnection` class now handles more send and receive errors: `ECONNABORTED`, `ECONNRESET`, `ENETRESET`, and `ETIMEDOUT`.
- Fixed setup.py to not include superfluous packages.

Development

- Added two exceptions: `BoofuzzTargetConnectionReset` and `BoofuzzTargetConnectionAborted`.
- These two exceptions are handled in `sessions.py` and may be thrown by any `ITargetConnection` implementation.

4.8.18 0.0.5

Fixes

- Boofuzz now properly reports crashes detected by the process monitor. It was calling log_info instead of log_fail.
- Boofuzz will no longer crash, but will rather give a helpful error message, if the target refuses socket connections.
- Add utils/crash_binning.py to boofuzz/utils, avoiding import errors.
- Fix procmon argument processing bug.
- Fix typos in INSTALL.rst.

4.8.19 0.0.4

- Add Gitter badge to README.
- Add default sleep_time and fuzz_data_logger for Session to simplify boilerplate.

4.8.20 0.0.3

- Fixed deployment from 0.0.2.
- Simplify CONTRIBUTING.rst for automated deployment.
- tox no longer runs entirely as sudo. The sudo has been moved into tox.ini and is more fine-grained.
- Reduced default `Session.__init__ restart_sleep_time` from 5 minutes to 5 seconds.

4.8.21 0.0.2

Continuous deployment with Travis.

Development

- Added build and PyPI badges.
- Added CONTRIBUTING.rst.
- check-manifest now runs in automated build.
- Travis now deploys to PyPI!

4.8.22 0.0.1-dev5

Development

- Tests now run on tox.
- Added Google Groups and Twitter link.

4.8.23 0.0.1-dev4

Fixes

- Missing property setters in `boofuzz.request.Request` now implemented.
- Unit tests now pass on Windows.
- Fixed wheel build issue; boofuzz subpackages were missing.

4.8.24 0.0.1-dev3

Fixes

- Session constructor param `session_filename` is now optional.

4.8.25 0.0.1-dev2

New features

- Now on PyPI! `pip install boofuzz`
- API is now centralized so all classes are available at top level `boofuzz.*`
  - This makes it way easier to use. Everything can be used like `boofuzz.MyClass` instead of `boofuzz.my_file.MyClass`.
- Added `EzOutletReset` class to support restarting devices using an ezOutlet EZ-11b.

Backwards-incompatible

- Target now only takes an `ITargetConnection`. This separates responsibilities and makes our code more flexible with different kinds of connections.

Fixes

- Bugs fixed:
  - `helpers.udp_checksum` was failing with oversized messages.
  - Missing install requirements.
  - Grammar and spelling.
  - `setup.py` was previously installing around five mostly unwanted packages. Fixed.
  - Removed deprecated unit tests.

  - Removed overly broad exception handling in Session.
  - `Checksum.render()` for UDP was not handling dependencies properly.

Back-end Improvements

This section took the most work. It has the least visible impact, but all of the refactors enable new features, fixes, and unit tests.

- Primitives and Blocks:
  - Created `IFuzzable` which properly defines interface for `Block`, `Request`, and all `BasePrimitive` classes.
  - Made effectively private members actually private.
  - Eliminated `exhaust()` function. It was used only once and was primarily a convoluted break statement. Now it's gone. :)
  - Split all block and primitive classes into separate files.
- Many Unit tests added.

Other

- Continuous integration with Travis is running!
- Doc organization improvements.
- Can now install with extras `[dev]`

4.8.26 Initial Development Release - 0.0.1-dev1

- Much easier install experience!
- Support for arbitrary communications mediums.
  - Added serial communications support.
  - Improved sockets to fuzz at Ethernet and IP layers.
- Extensible instrumentation/failure detection.
- Better recording of test data.
  - Records all sent and received data
  - Records errors in human-readable format, in same place as sent/received data.
- Improved functionality in checksum blocks.
- Self-referential size and checksum blocks now work.
- `post_send` callbacks can now check replies and log failures.
- Far fewer bugs.
- Numerous refactors within framework code.

CHAPTER

FIVE

CONTRIBUTIONS

Pull requests are welcome, as boofuzz is actively maintained (at the time of this writing ;)). See *Contributing*.

CHAPTER

SIX

COMMUNITY

For questions that take the form of "How do I… with boofuzz?" or "I got this error with boofuzz, why?", consider posting your question on Stack Overflow. Make sure to use the `fuzzing` tag.

If you've found a bug, or have an idea/suggestion/request, file an issue here on GitHub.

For other questions, check out boofuzz on gitter or Google Groups.

For updates, follow @b00fuzz on Twitter.

CHAPTER

SEVEN

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

b

INDEX

F

G

H

I

L

M

N

O

P

R

S

T

U