tlsfuzzer Documentation

Release 0.0.0

Hubert Kario

Sep 29, 2021

Contents:

`tlsfuzzer` tests *SSL* and *TLS* implementations.

It allows for testing standards-compliance of a given implementation, testing for presence of known vulnerabilities as well as fuzzing of the *SSL* and *TLS* connections.

You can find ready to use scripts that test significant parts of *TLS* protocols in the source repository.

The testing of OpenSSL, GnuTLS, NSS, and other implementations commonly includes running tlsfuzzer test cases.

While tlsfuzzer doesn't support some features of *TLS*, it includes the most commonly used ones: *TLS* 1.2, *TLS* 1.3, *RSA* certificates, *ECDSA* certificates, *ECDHE* key exchange, client certificates, *AES-GCM*, Chacha20-Poly1305 ciphers, etc. See the issue tracker on GitHub to see wanted, but not yet implemented features.

CHAPTER 1

Quickstart

1.1 Installing dependencies

To execute tlsfuzzer test scripts you need a python environment. This framework supports all versions of python since 2.6 except 3.0, 3.1, and 3.2. Check the Travis CI to see explicitly tested environments.

Python supports installing modules system-wide, to the user directory, or to a virtual environment. With `tlsfuzzer` dependencies you can use either option, though some work better than others.

Note: Execute all example commands in the root directory of tlsfuzzer repository checkout.

Hint: If you plan to develop, not just use tlsfuzzer, use the instructions in the *Installation* chapter. If you want to try several scripts before installing full development environment, for swift clean up, use the virtual environment installation method.

1.1.1 System wide installation

Installation of modules system-wide allows for easy execution of scripts later. This does "pollute" the system and conflicts with python modules managed by the OS package manager though. It also requires administrative privileges on the system. You should use this approach if you plan to keep using tlsfuzzer for a long time.

To install all dependencies execute as root:

```
pip install -r requirements.txt
```

Warning: Different versions of python keep their modules separate, as such, installing packages with `pip` from Python 2.7 doesn't make them available for Python 3.7 and vice versa.

1.1.2 User directory installation

If you don't have administrative privileges on the system, you can install python modules to your local home directory. This doesn't make them usable for other users of the system. Unlike the virtual environment approach, it does make running with wrong python environment less probable.

To install all dependencies to user directory execute:

```
pip install --user -r requirements.txt
```

For Python 3.7 this places the modules to the `~/.local/lib/python3.7/site-packages/` directory. For Python 2.7 this places the modules to the `~/.local/lib/python2.7/site-packages/` directory.

1.1.3 Virtual environment installation

You can find detailed description of Python virtual environments in the official documentation. Deleting a virtual environment doesn't influence anything outside of it, making it safe to do after you don't need it.

To create a virtual environment in a new directory, for example `~/tlsfuzzer-env`, execute:

```
python -m venv ~/tlsfuzzer-env
```

To install all dependencies in that virtual environment execute:

```
~/tlsfuzzer-env/bin/pip install -r requirements.txt
```

Note: When you use virtual environments you must specify the `python` executable from the virtual environment, not the system-wide one. Use `~/tlsfuzzer-env/bin/python` instead of `python` to execute the test scripts in following examples. You can also "activate" an environment to make `python` and `pip` point to commands from the virtual environment, this modifies only the current session though. To do that execute `source ~/tlsfuzzer-env/bin/activate`.

1.2 Starting an OpenSSL server

To have a server to test against you can use OpenSSL. Example below shows how to setup a configuration with a self-signed certificate. You can execute the scripts against any network-accessible server, if you have one already running, you can skip this part.

1.2.1 Generate certificates

Most test cases require a server configured with a certificate (the ones that require more complex *PKIX* setup print it when executed).

To create a simple self-signed certificate and key, execute the following OpenSSL command:

```
openssl req -x509 -newkey rsa -keyout /tmp/localhost.key \
-out /tmp/localhost.crt -subj /CN=localhost -nodes -batch \
-days 3650
```

1.2.2 Start the server

Once you have a key and a certificate, you can use them to configure a test server with support for minimal subset of HTTP:

```
openssl s_server -key /tmp/localhost.key -cert /tmp/localhost.crt -www
```

1.3 Executing a test case

With a *TLS* server available, you can start executing test cases against it.

To verify that a server supports *TLS* 1.2 or earlier, you can use the `test-conversation.py` script.

To execute the script against a server running on `localhost` on port 4433, as it's set-up in the preceding OpenSSL example, execute the following command in the checkout of tlsfuzzer repository:

```
PYTHONPATH=. python scripts/test-conversation.py
```

This command should provide the following output if everything went fine:

```
sanity ...
OK

sanity ...
OK

Basic conversation script; check basic communication with typical
cipher, TLS 1.2 or earlier and RSA key exchange (or (EC)DHE if
-d option is used)

version: 4

Test end
successful: 2
failed: 0
```

All the test scripts support at least `--help` option. For this script it will provide the following information:

```
Usage: <script-name> [-h hostname] [-p port] [[probe-name] ...]
 -h hostname    name of the host to run the test against
                localhost by default
 -p port        port number to use for connection, 4433 by default
 probe-name     if present, will run only the probes with given
                names and not all of them, e.g "sanity"
 -e probe-name  exclude the probe from the list of the ones run
                may be specified multiple times
 -n num         only run `num` random tests instead of a full set
                ("sanity" tests are always executed)
 -d             negotiate (EC)DHE instead of RSA key exchange
 --help         this message
```

Almost all scripts support this set of command line options.

Executing a test case to verify *TLS* 1.3 support works similar:

```
PYTHONPATH=. python scripts/test-tls13-conversation.py
```

This produces similar output:

```
sanity ...
OK

sanity ...
OK

Basic communication test with TLS 1.3 server
Check if communication with typical group and cipher works with
the TLS 1.3 server.

version: 2

Test end
successful: 2
failed: 0
```

Similarly to the *TLS* 1.2 script, this one supports a set of options:

```
Usage: <script-name> [-h hostname] [-p port] [[probe-name] ...]
 -h hostname    name of the host to run the test against
                localhost by default
 -p port        port number to use for connection, 4433 by default
 probe-name     if present, will run only the probes with given
                names and not all of them, e.g "sanity"
 -e probe-name  exclude the probe from the list of the ones run
                may be specified multiple times
 -n num         only run `num` random tests instead of a full set
                ("sanity" tests are always executed)
 --help         this message
```

As cryptographic parameter negotiation happens differently in *TLS* 1.3 than it does in *TLS* 1.2, the *TLS* 1.3 scripts generally don't support the `-d` option.

Note: When a particular test case in the script observes an expected behaviour it prints an "OK" status, if all test cases in a test script do that, the script passes. Expected behaviour doesn't mean a successful connection. Negative test cases *expect* a failed *TLS* handshake or a particular kind of connection abortion.

CHAPTER 2

Installation

The project is set up so that installation of it or dependencies is not necessary.

Installing the dependencies will make handling all dependencies easier though.

(More complete instructions are in CONTRIBUTING.md and USAGE.md files.)

2.1 `pip`

Because the tlsfuzzer is developed in lock-step with tlslite-ng, only the newest releases of the latter are expected to work. That means either alpha or beta versions of tlslite-ng.

To install the latest version tested use `pip`:

```
pip install -r requirements.txt
```

2.2 Using source directly

If the dependencies of tlslite-ng are already installed, the only part of tlslite-ng necessary for tlsfuzzer to work, is the `tlslite` module. As such, it's possible to just link the `tlslite` directory in a checkout of tlslite-ng project inside the checkout of tlsfuzzer.

If both tlsfuzzer and tlslite-ng have been cloned to the same directory, it's enough to execute the following command inside the tlsfuzzer directory:

```
ln -s ../tlslite-ng/tlslite tlslite
```

2.3 Virtual environments

If you would like to install tlslite-ng or its dependencies, but not affect the general system, or even your personal python packages, it's possible to use virtual environments.

CHAPTER 3

Theory

By cooperating with each other, the record layer protocol, the handshake protocol, the alert protocol, and the application data protocol create the *TLS* protocol.

By working together, they establish a connection that provides an integrity-protected tunnel or socket. The connection, if negotiated, also include authentication of server, or both server and client. Most connections also provide encryption of the data travelling in the tunnel.

By modifying the messages sent, tester can check if the other side establishes the connection in expected circumstances. Tester can also check if an implementation aborts the connection on protocol violations, use of unimplemented, or turned off features. Same for the integrity, authentication, and encryption, by modifying the data sent, tester can verify if the other side implements the checks necessary to provide these properties.

3.1 TLS protocols

The record layer protocol provides the multiplexing capability to exchange the data from the other *TLS* protocols over the same TCP connection or socket.

3.1.1 Record layer

The record layer protocol uses records to transfer data belonging to a given upper level protocol. It provides a stream abstraction, just like a TCP connection. That means, the upper layer protocols can't depend on writes generating a particular number or size of records. The record layer can combine messages into a single record with other data of the same higher level protocol.

In particular, a record can have at most 2^{14} bytes of payload. To process bigger messages from higher level protocols (e.g. ClientHello) record layer fragments them and sends them in more than one record.

By extending the payload with the protocol type and size of the payload, the record layer provides multiplexing to the higher level protocols.

Record layer protects the integrity of exchanged data and, optionally, encrypts and decrypts data. It uses keys and ciphers negotiated by the handshake protocol to do that.

3.1.2 Handshake protocol

Handshake protocol establishes the keys used in the connection and, optionally, the identities of the server or server and client.

Similarly to record layer, handshake protocol messages also include the payload type and payload size.

Unlike the record layer, handshake protocol limits the size of messages to $2^{24} - 1$ bytes. Handshake protocol also forbids fragmenting or combining of the messages.

3.1.3 Application data protocol

Application data protocol encapsulates the data provided to *TLS* so that it can travel in the same connection as messages internal to *TLS*. It serves as a content type to the record layer protocol.

But just like other protocols travelling over record layer, it can't depend on specific fragmentation of writes to the other side.

3.1.4 Alert protocol

Alert protocol provides signalling of error conditions or unmet expectations to the other side of the connection. When messaging non-fatal errors, in some cases, the connection can continue even after their exchange.

An alert message consists of two bytes.

3.2 Testing process

The basic testing scenarios focus on the so called "happy path": verifying that everything works when nothing unexpected occurred. While testing for support of features needs to use this kind of approach, negative test cases must use malformed or unexpected messages, especially in security protocols. Correct handling of unexpected situations provides the security.

The *TLS* specification requires strict verification of message format from the parsers. It also describes precisely the expected contents of majority of exchanged fields—encryption or integrity protection of messages allows for one valid and correct formatting of messages or records, for a given set of keys. The specification includes also information on error handling, it describes the expected alert messages for given error conditions.

This allows the tests to send either malformed or inconsistent messages and check for specified alerts to verify if the other side of the connection performed the expected error checking.

Note: Fuzzers generally don't operate in this way. Typical fuzzers feed the system under test (*SUT*) with lots of random or semi-random inputs and check if the *SUT* doesn't crash, use uninitialised memory or invokes some other undefined behaviour. While tlsfuzzer can generate this kind of tests, included scripts don't do it—they focus on checking if the server behaves as expected, even when they use random data for it.

3.2.1 Checking alerts

Given that the guiding RFCs allow for *not* sending the alerts at all, one could argue that checking both reception of alerts and the included error codes in them to be undue carefulness.

Actually though exploitation of security vulnerabilities thanks to the different error codes returned for different errors detected has a long history. When returned errors depend on secret data, unknown to attacker, that may lead to

decryption oracles or other side-channel attacks. The standards do take this into account, which makes standard-compliant behaviour the "known good" behaviour.

Consistent and standards-compliant errors also make debugging of interoperability issues easier. Alert description points to the reason of rejection: a certificate issue, a malformed message, a message inconsistent with other messages, etc.

Consistent and correct alerts also allow pushing those errors higher in the stack—if user-level application can depend on particular meaning of errors it can provide more correct and relevant errors to the user.

To confidently test for security vulnerabilities across different implementations, the implementations must behave in consistent, or at least similar ways. When they do, tlsfuzzer can reuse a single verification script to test them.

When test doesn't have an easy insight into the process serving *TLS*, getting the alert instead of connection close allows for at least basic verification if the *SUT* didn't crash but handled the error.

Sharing of general test suites has the same limitations as sharing of security test scripts. If different implementations exhibit the same behaviour, they can share the same test suite, in turn reducing effort necessary to develop new implementations or extend existing implementations with new features.

Last, but not least, particular way of handling errors provides a strong signal for fingerprinting (identifying) the implementation used. As alert descriptions returned by an implementation don't depend on implementation configuration, the fingerprints don't either, making them robust—hard to masquerade one implementation for another (with some exceptions, like in case the server doesn't parse extensions from turned-off features).

CHAPTER 4

Simple test creation

Network servers use connection timeouts to drop stalled or unused connections. For some that happens in a minute or two, for others in seconds. Thus, robust test cases require automation. tlsfuzzer achieves it through a runner that executes decision graphs.

The test scripts included in `scripts/` directory build the decision graph necessary for testing different scenarios. After building a graph, the runner executes it and provides a test result (by raising an exception in case of errors). The example below builds a single graph and executes it.

4.1 Building decision graph

To exchange *TLS* messages the script needs to establish a *TCP* connection. `Connect` takes the server's hostname and a port number to do that:

```
from tlsfuzzer.messages import Connect
root_node = Connect("localhost", 4433)
node = root_node
```

4.1.1 ClientHello

Next step requires sending the first message of the *TLS* handshake: the ClientHello. This node requires at least two parameters: the list of cipher suites and a dictionary of extensions.

`CipherSuite` class lists cipher suites supported by the project or defined by *IETF*. To establish a connection with ones that use *ECDHE* key exchange and most commonly used *AES* ciphers, define the following list:

```
from tlslite.constants import CipherSuite
ciphers = [
    CipherSuite.TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256,
    CipherSuite.TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256,
    CipherSuite.TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA,
```

(continues on next page)

(continued from previous page)

```
    CipherSuite.TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA
]
```

Connections that use *ECDHE* key exchange need to advertise to the server the elliptic curves supported by the client. Those advertisements travel inside extensions.

ClientHelloGenerator requires passing the extensions as a `dict` or similar object:

```
extensions = {}
```

`GroupName` class lists the groups defined for *TLS*. To use the two most common ones write:

```
from tlslite.constants import GroupName
groups = [
    GroupName.secp256r1,
    GroupName.x25519
]
```

To send that list to the server, package it into a *TLS* extension object. That happens in `SupportedGroupsExtension`:

```
from tlslite.extensions import SupportedGroupsExtension
from tlslite.constants import ExtensionType
groups_ext = SupportedGroupsExtension().create(groups)
extensions[ExtensionType.supported_groups] = groups_ext
```

Since servers sign *ECDHE* key exchange, clients need to advertise the signature algorithms they support. That happens in `SignatureAlgorithmsExtension` object.

To build a list of most common signature algorithms include:

```
from tlslite.constants import (
    SignatureScheme,
    HashAlgorithm,
    SignatureAlgorithm
)
sig_algs = [
    SignatureScheme.ecdsa_secp521r1_sha512,
    SignatureScheme.ecdsa_secp384r1_sha384,
    SignatureScheme.ecdsa_secp256r1_sha256,
    SignatureScheme.rsa_pss_pss_sha512,
    SignatureScheme.rsa_pss_pss_sha384,
    SignatureScheme.rsa_pss_pss_sha256,
    SignatureScheme.rsa_pss_rsae_sha512,
    SignatureScheme.rsa_pss_rsae_sha384,
    SignatureScheme.rsa_pss_rsae_sha256,
    SignatureScheme.rsa_pkcs1_sha512,
    SignatureScheme.rsa_pkcs1_sha384,
    SignatureScheme.rsa_pkcs1_sha256,
    (HashAlgorithm.sha1, SignatureAlgorithm.ecdsa),
    SignatureScheme.rsa_pkcs1_sha1
]
```

Then to convert it to an extension include:

```
from tlslite.extensions import SignatureAlgorithmsExtension
sig_algs_ext = SignatureAlgorithmsExtension().create(sig_algs)
extensions[ExtensionType.signature_algorithms] = sig_algs_ext
```

Clients need to advertise support for safe renegotiation, even if they don't support renegotiation or intend to perform it. To advertise it, send an empty `renegotiation_info` extension, like so:

```
from tlslite.extensions import RenegotiationInfoExtension
renego_ext = RenegotiationInfoExtension().create(b'')
extensions[ExtensionType.renegotiation_info] = renego_ext
```

After preparing all extensions, create the ClientHello object and attach it to the decision graph:

```
from tlsfuzzer.messages import ClientHelloGenerator
node = node.add_child(ClientHelloGenerator(ciphers, extensions=extensions))
```

4.1.2 Server reply

Nodes responsible for processing server response use values specified in ClientHello as defaults, as such, they don't need any parameters:

```
from tlsfuzzer.expect import (
    ExpectServerHello, ExpectCertificate, ExpectServerKeyExchange,
    ExpectServerHelloDone
)
node = node.add_child(ExpectServerHello())
node = node.add_child(ExpectCertificate())
node = node.add_child(ExpectServerKeyExchange())
node = node.add_child(ExpectServerHelloDone())
```

4.1.3 Client's key share and finish

Since ServerKeyExchange message includes the group selected by the server, the client can generate its own key share and send it back.

Again, as the client nodes look at exchanged messages in the connection, they don't need any parameters:

```
from tlsfuzzer.messages import (
    ClientKeyExchangeGenerator,
    ChangeCipherSpecGenerator,
    FinishedGenerator
)
node = node.add_child(ClientKeyExchangeGenerator())
node = node.add_child(ChangeCipherSpecGenerator())
node = node.add_child(FinishedGenerator())
```

Note: *`ChangeCipherSpecGenerator`* reconfigures the record layer to use encryption for sending the following messages.

4.1.4 Server's finish

Server accepts the handshake as successful by sending its own ChangeCipherSpec and Finished, so the script needs to expect them:

```
from tlsfuzzer.expect import (
    ExpectChangeCipherSpec,
    ExpectFinished
)
node = node.add_child(ExpectChangeCipherSpec())
node = node.add_child(ExpectFinished())
```

Note: *ExpectChangeCipherSpec()* reconfigures the record layer to use encryption for receiving the following messages.

4.1.5 Application data

What happens after the handshake depends on the application protocol that uses *TLS*. To perform a single `GET` with HTTP 1.0, use the following:

```
from tlsfuzzer.messages import ApplicationDataGenerator
from tlsfuzzer.expect import ExpectApplicationData
request = b"GET / HTTP/1.0\r\n\r\n"
node = node.add_child(ApplicationDataGenerator(request))
node = node.add_child(ExpectApplicationData())
```

4.1.6 Closing the connection (alternatives in decision graphs)

To handle slight differences between different ways that servers behave, the framework allows specifying alternatives for the expected messages. Since some servers reply with `close_notify` Alert to client's `close_notify` while others close the connection instantly, the script needs to reflect that.

Tip: If you want to verify that the server *does* send an Alert before closing the connection, don't use the alternative mechanism. Rather specify the expected behaviour as connection close after Alert, without the use of `next_sibling`.

To trigger connection close send the alert:

```
from tlsfuzzer.messages import AlertGenerator
from tlslite.constants import AlertLevel, AlertDescription
node = node.add_child(AlertGenerator(AlertLevel.warning,
                                     AlertDescription.close_notify))
```

Nodes include alternative paths in the `next_sibling` field. To specify that the script should expect connection close with or without an Alert before connection close, use the following code:

```
from tlsfuzzer.expect import ExpectAlert, ExpectClose

node = node.add_child(ExpectAlert())
node.next_sibling = ExpectClose()
node.add_child(ExpectClose())
```

With no more nodes in the graph, the runner closes the connection and ignores any data in buffers. *ExpectClose* instead verifies that server didn't send any messages before closing the socket.

You can read more about alternatives in the *Decision graph* chapter.

4.2 Executing decision graphs

If you tried to execute this example script now, nothing would happen. To actually connect to a server and exchange messages, the runner needs to execute the decision graph.

As an argument the runner takes the root of the decision graph. In case of unmet expectations (*TCP* connection failure, misbehaviour by the server, etc.) the runner raises an exception.

To prepare it execute:

```
from tlsfuzzer.runner import Runner
runner = Runner(root_node)
```

To execute the decision graph:

```
runner.run()
```

4.3 Source code of the example

You can find this example with better formatting, help message, command line option parsing, and support for *RSA* key exchange in scripts/test-conversation.py. If you want to contribute test cases to this project you should use this file as a template for *TLS* 1.2 or earlier test cases. For *TLS* 1.3 test cases you should use scripts/test-tls13-conversation.py.

With no clean-up this example looks like this:

```
from tlsfuzzer.messages import Connect
root_node = Connect("localhost", 4433)
node = root_node

from tlslite.constants import CipherSuite
ciphers = [
    CipherSuite.TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256,
    CipherSuite.TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256,
    CipherSuite.TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA,
    CipherSuite.TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA
]

extensions = {}

from tlslite.constants import GroupName
groups = [
    GroupName.secp256r1,
    GroupName.x25519
]

from tlslite.extensions import SupportedGroupsExtension
from tlslite.constants import ExtensionType
groups_ext = SupportedGroupsExtension().create(groups)
extensions[ExtensionType.supported_groups] = groups_ext
```

(continues on next page)

(continued from previous page)

```
from tlslite.constants import (
    SignatureScheme,
    HashAlgorithm,
    SignatureAlgorithm
)
sig_algs = [
    SignatureScheme.ecdsa_secp521r1_sha512,
    SignatureScheme.ecdsa_secp384r1_sha384,
    SignatureScheme.ecdsa_secp256r1_sha256,
    SignatureScheme.rsa_pss_pss_sha512,
    SignatureScheme.rsa_pss_pss_sha384,
    SignatureScheme.rsa_pss_pss_sha256,
    SignatureScheme.rsa_pss_rsae_sha512,
    SignatureScheme.rsa_pss_rsae_sha384,
    SignatureScheme.rsa_pss_rsae_sha256,
    SignatureScheme.rsa_pkcs1_sha512,
    SignatureScheme.rsa_pkcs1_sha384,
    SignatureScheme.rsa_pkcs1_sha256,
    (HashAlgorithm.sha1, SignatureAlgorithm.ecdsa),
    SignatureScheme.rsa_pkcs1_sha1
]

from tlslite.extensions import SignatureAlgorithmsExtension
sig_algs_ext = SignatureAlgorithmsExtension().create(sig_algs)
extensions[ExtensionType.signature_algorithms] = sig_algs_ext

from tlslite.extensions import RenegotiationInfoExtension
renego_ext = RenegotiationInfoExtension().create(b'')
extensions[ExtensionType.renegotiation_info] = renego_ext

from tlsfuzzer.messages import ClientHelloGenerator
node = node.add_child(ClientHelloGenerator(ciphers, extensions=extensions))

from tlsfuzzer.expect import (
    ExpectServerHello, ExpectCertificate, ExpectServerKeyExchange,
    ExpectServerHelloDone
)
node = node.add_child(ExpectServerHello())
node = node.add_child(ExpectCertificate())
node = node.add_child(ExpectServerKeyExchange())
node = node.add_child(ExpectServerHelloDone())

from tlsfuzzer.messages import (
    ClientKeyExchangeGenerator,
    ChangeCipherSpecGenerator,
    FinishedGenerator
)
node = node.add_child(ClientKeyExchangeGenerator())
node = node.add_child(ChangeCipherSpecGenerator())
node = node.add_child(FinishedGenerator())

from tlsfuzzer.expect import (
    ExpectChangeCipherSpec,
    ExpectFinished
)
node = node.add_child(ExpectChangeCipherSpec())
```

(continues on next page)

(continued from previous page)

```
node = node.add_child(ExpectFinished())

from tlsfuzzer.messages import ApplicationDataGenerator
from tlsfuzzer.expect import ExpectApplicationData
request = b"GET / HTTP/1.0\r\n\r\n"
node = node.add_child(ApplicationDataGenerator(request))
node = node.add_child(ExpectApplicationData())

from tlsfuzzer.messages import AlertGenerator
from tlslite.constants import AlertLevel, AlertDescription
node = node.add_child(AlertGenerator(AlertLevel.warning,
                                     AlertDescription.close_notify))

from tlsfuzzer.expect import ExpectAlert, ExpectClose

node = node.add_child(ExpectAlert())
node.next_sibling = ExpectClose()
node.add_child(ExpectClose())

from tlsfuzzer.runner import Runner
runner = Runner(root_node)

runner.run()
```

CHAPTER 5

Decision graph

While this documentation calls the structure traversed by the runner a "decision graph," as it can contain loops, it's more precisely described as a directed graph. Older parts of this documentation and object names refer to this structure as a "decision tree"—this name reflects the most common use case in the bundled tests, not the most complex supported one.

5.1 Node fields

A decision graph node, a *TreeNode*, has two pointers, to a `child` and to a `next_sibling`. On initialisation nodes set them to `None`.

5.1.1 Child nodes

When a node matches received message and processes it without errors, *Runner* continues execution by switching to the child.

If `child` points to `None`, runner closes open connections and ends execution.

To create loops the `child` can point to itself or nodes that point to it, either directly or transitively. You need to use this mechanism to allow receiving arbitrary number of messages.

5.1.2 Sibling nodes

The runner uses nodes pointed to by `next_sibling` when received message doesn't match the current node. When sending messages, runner looks into `next_sibling` when connection got closed.

You can use this mechanism to either break out of loops or to define alternatives in execution.

5.2 Advanced decision graph structures

As mentioned before, the decision graph allows for non-linear relationship between nodes.

5.2.1 Loops

Test case runner in tlsfuzzer can accept arbitrary number of messages if the node points to itself as its child.

For example, to accept zero or more NewSessionTicket messages in *TLS* 1.3 connection, the script needs to include the following code:

```
cycle = ExpectNewSessionTicket()
node = node.add_child(cycle)
node.add_child(cycle)
```

Servers that send the NewSessionTicket after Finished and before any other messages, require the preceding code after *ExpectFinished*. That handles OpenSSL-using servers and others that behave similarly.

Note: *TLS* standard does allow sending NewSessionTicket messages at arbitrary times after Finished.

Write the following code to make the runner finish the loop once an ApplicationData message is received:

```
node.next_sibling = ExpectApplicationData()
node = node.next_sibling
```

Tip: If you want to accept arbitrary number of NewSessionTicket messages, but no fewer than a specified amount, add more *ExpectNewSessionTicket* nodes before the loop to ensure that server sends them.

You can find a working example of this code in test-tls13-conversation.py.

5.2.2 Alternatives

Servers configured with client certificate based authentication send CertificateRequest message. For a script to interoperate with such servers it needs to expect that message. If a client receives it, it needs to reply with a Certificate message, even if it doesn't have a certificate (it sends an empty message then). Since a node doesn't have a limit on the number of parent nodes, script can specify a branch to handle such connections.

Start with specifying the exceptional path, save reference to the fork point:

```
node = node.add_child(ExpectCertificateRequest())
fork = node
node = node.add_child(ExpectServerHelloDone())
node = node.add_child(CertificateGenerator())
```

Then specify the usual path, for servers that don't ask for client certificates:

```
fork.next_sibling = ExpectServerHelloDone()
```

In both handshake scenarios the client sends ClientKeyExchange message, this joins the paths:

```
join = ClientKeyExchangeGenerator()
# join regular path:
fork.next_sibling.add_child(join)
# join CR path:
node = node.add_child(join)
```

After that, handshake continues as usual with ChangeCipherSpec, Finished, etc.

Note: When specifying alternative messages, you must take care not to allow message exchanges forbidden by the standards. Place all the messages that depend on the branch in the branch to ensure that (but check if using a command line switch to build different graphs doesn't lead to simpler test scripts).

You can find a working example of this code in test-fuzzed-plaintext.py.

5.2.3 Error handling

If you want to allow the server to abort connection while *sending* data, use the sibling mechanism too.

To allow the server to close the connection while writing to it, specify the *ExpectClose* as sibling of the node:

```
node = node.add_child(CertificateVerifyGenerator(private_key))
node.next_sibling = ExpectClose()
node = node.add_child(ChangeCipherSpecGenerator())
node.next_sibling = ExpectClose()
node = node.add_child(FinishedGenerator())
node.next_sibling = ExpectClose()
```

Use *ExpectAlert* the same way.

Note: Runner supports only *ExpectAlert* and *ExpectClose* as siblings of generator nodes. Since connection close triggers this path, you can read only already buffered messages.

You can find a working example of this code in test-certificate-verify-malformed-sig.py.

CHAPTER 6

Message manipulation

Tlsfuzzer provides facilities to modify messages and records before sending, use them to create malformed messages. You can apply the modifiers on generator nodes, the ones that send messages to the peer.

6.1 Custom message generation

Tlsfuzzer provides support for sending arbitrary messages over established connections. It provides two nodes to achieve it: one to send messages unencrypted and one to send them using the current connection status.

6.1.1 Creating unencrypted messages

To send a record with a specific payload and type, irrespective of active encryption or negotiated fragmentation, use *PlaintextMessageGenerator*. It accepts two parameters to specify data sent to the other peer (`content_type` and `data`) as well as one used for debugging: `description`, printed when sending of the message failed.

Note: As it skips all the usual message processing steps, it also doesn't update handshake hashes so values calculated for Finished and connection secrets in *TLS* 1.3 won't match expected ones.

For example, to send an empty ClientHello message, write:

```
node = node.add_child(PlaintextMessageGenerator(
    ContentType.handshake,
    bytearray(b'\x01\x00\x00\x00')))
```

You can find a usage example in: test-aesccm.py.

Tip: If you want to send an otherwise valid message, only as plaintext, not encrypted, see the *Clearing encryption settings* section.

To write directly to the socket, without record layer encapsulation, use the *RawSocketWriteGenerator*. It accepts two parameters, one to specify the data to write and another, optional, used for debugging, the `description`.

6.1.2 Creating arbitrary messages

To send messages with a specific payload and type, while using encryption and record layer fragmentation, use *RawMessageGenerator*.

It accepts two parameters that specify data sent to the other side (`content_type` and `data`) and one that stores message to print if processing of the message fails: `description`.

For example, to send an empty Finished message, write:

```
node = node.add_child(RawMessageGenerator(
    ContentType.handshake,
    bytearray(b'\x14\x00\x00\x00')
```

You can find a usage example in: test-invalid-content-type.py.

6.2 Modifying messages

Tlsfuzzer supports applying two operations to sent messages: modifying length and modifying contents of specific bytes.

6.2.1 Modifying length

Handshake messages include an internal header that identifies the message type and message length. Two methods can change their payload while modifying the header to match.

The *pad_handshake()* function adds data at the end of payload. The `size` param specifies how many bytes and the `pad_byte` parameter specifies the value of the added bytes.

In the other calling convention, it accepts literal bytes to add to the payload by using the `pad` keyword argument.

For example, to add 10 bytes of value 0 at the end of ClientHello, write:

```
ciphers = [CipherSuite.TLS_RSA_WITH_AES_128_CBC_SHA]
exts = {ExtensionType.renegotiation_info: None}
msg_gen = ClientHelloGenerator(cipihers, extensions=exts)
node = node.add_child(pad_handshake(msg_gen, 10))
```

You can find a usage example in: test-truncating-of-client-hello.py.

If you want to remove bytes from the end of a message, you can either specify a negative `size` or use the *truncate_handshake()* function.

Note: The sender can format ClientHello in two ways: with and without extensions. A ClientHello with an empty list of extensions differs from one without extensions by two zero bytes (they encode the length of the extensions). Thus adding 2 zero bytes to an extensions-less ClientHello or removing enough bytes from a ClientHello with extensions to turn it into one without extensions can cause the *pad_handshake()* to create a well-formed message, despite modifying it.

6.2.2 Modifying content

The *fuzz_message()* supports changing arbitrary parts of sent messages.

Both optional parameters of the function, `substitutions` and `xors` expect a dictionary as value. The keys of the dictionary specify the bytes to change. To specify the bytes counting from the end of the message use negative numbers.

For example, to change the type of a ClientHello message to that of ServerHello use the following code:

```
ciphers = [CipherSuite.TLS_RSA_WITH_AES_128_CBC_SHA]
exts = {ExtensionType.renegotiation_info: None}
msg_gen = ClientHelloGenerator(cipihers, extensions=exts)
node = node.add_child(fuzz_message(msg_gen,
                                   {0: HandshakeType.server_hello}))
```

You can find a usage example in: test-invalid-client-hello.py.

6.3 Modifying records

The *TLS* protocol specifies four types of encrypted records: ones that use stream encryption, ones that use block encryption in *MAC* then encrypt mode, ones that use block encryption in encrypt then *MAC* mode, and ones that use *AEAD* ciphers. Each of them behaves differently on the record layer level, thus modifying the intermediate ciphertext requires the use of different functions.

6.3.1 Fuzzing the MAC

To change the authentication tag you need to use different functions depending on which cipher suite and extensions have been negotiated.

For ciphers that use *HMAC* you can change the authentication tag using the *fuzz_mac()* function.

Note: *fuzz_mac()* works with stream ciphers and block ciphers in *CBC* mode only. It doesn't work for SSLv2 connections though.

You use *fuzz_mac()* the same way as you use *fuzz_message()*: pass the message to change as the first argument and use the other two to specify the bytes to either xor or substitute.

Use the following code to invert the first and last bit of the :term'HMAC' in a record with a Finished message:

```
msg_gen = FinishedGenerator()
xors = {0: 0x80, -1: 0x01}
node = node.add_child(fuzz_mac(msg_gen, xors=xors))
```

You can find a usage example in: test-fuzzed-MAC.py.

Since both *AEAD* cipher suites and *CBC* cipher suites in "encrypt then *MAC*" mode don't encrypt the authentication tag, you can use the *fuzz_encrypted_message()* function to change it. As it allows modification of any part of encrypted message, not just the tag, you need to know the size of the authentication tag to change the first byte of it though.

Hint: *AES-CCM8* uses tags 8 bytes long. *AES-GCM*, Chacha20-Poly1305, *AES-CCM* and MD5-HMAC use tags 16 bytes long. SHA1-HMAC uses tags 20 bytes long. SHA256-HMAC uses tags 32 bytes long. SHA384-HMAC uses

tags 48 bytes long

Use the following code to invert the first and last bit of authentication tag in a record with a Finished message in an *AES-GCM* connection:

```
msg_gen = FinishedGenerator()
xors = {-17: 0x80, -1: 0x01}
node = node.add_child(fuzz_encrypted_message(msg_gen, xors=xors))
```

You can find a usage example in: test-chacha20.py.

Tlsfuzzer can go as far as changing the whole plaintext right before encryption, this can change the *HMAC* for *CBC* mode ciphers working in "encrypt then *MAC*" mode. Use the *replace_plaintext()* function for that.

Hint: The length of the replacement plaintext must be a multiple of cipher's block size: 8 bytes for 3DES and 16 bytes for other ciphers.

For example, to create a record with a plaintext with all bytes of the *IV* set to 1 (assuming *AES* cipher), all bytes of the payload set to 2, all bytes of the authentication tag set to 3 (assuming SHA1-*HMAC*), and a zero-length padding, use the following code:

```
iv_bytes = bytearray([1]*16)
payload_bytes = bytearray([2]*11)
mac_bytes = bytearray([3]*20)
pad_bytes = bytearray(b'\x00')
new_plaintext = iv_bytes + payload_bytes + mac_bytes + pad_bytes
assert len(new_plaintext) % 16 == 0
msg_gen = FinishedGenerator()
node = node.add_child(replace_plaintext(msg_gen, new_plaintext))
```

You can find a usage example in: test-fuzzed-plaintext.py.

While you can use the *fuzz_plaintext()* function to change the *MAC*, you need to know the length of padding to know where *MAC* begins and ends in the plaintext.

6.3.2 Fuzzing the padding

The *CBC* mode ciphers require input with length that's a multiple of the cipher block size. Since stream ciphers and *AEAD* ciphers dont't require that, *TLS* 1.2 and earlier doesn't define padding for them.

As a single byte encodes the length of the padding, 255 bytes is the max length (256 bytes including the byte encoding length).

TLS 1.3 defines padding differently, it combines it with content type specification for record payload, thus the max record length (2^{14} or 16384 bytes) defines max padding.

The *fuzz_padding()* function can change the padding used by *CBC* cipher suites.

For example, to negate the last byte of padding of a record with Finished message (while ensuring non-zero length padding), use the following code:

```
msg_gen = FinishedGenerator()
node = node.add_child(fuzz_padding(msg_gen, min_length=1,
                                   xors={-2: 0xff}))
```

You can find a usage example in: test-fuzzed-padding.py.

While you can use the *fuzz_plaintext()* function to change the padding, it doesn't support specifying the min length for the padding.

6.3.3 TLS 1.3 padding length

tlsfuzzer supports changing the padding in sent records through a callback mechanism. The *SetPaddingCallback* node sets the callback for calculating the padding size. It includes two factory methods and one ready to use callback.

For example, to make all records send max supported padding in the connection, use the following code:

```
node = node.add_child(
    SetPaddingCallback(SetPaddingCallback.fill_padding_cb))
```

You can find a usage example in: test-tls13-record-layer-limits.py.

6.3.4 Sending too big records

The *TLS* protocol specifies the max length of payload at 2^{14} bytes. To send records with larger payload use *SetMaxRecordSize* to increase that limit.

Note: This increases the max length of *payload*. With active encryption, records include *IV*, *MAC* and padding or *AEAD* tag, making them at least 16 bytes larger.

Warning: The *TLS* protocol specifies the length in record header as two bytes, as such, records larger than 2^{16}-1 or 65535 bytes have no physical representation and tlsfuzzer doesn't support sending them. *IV*, padding and authentication tag increase the size of record compared to the payload by at least 16 bytes and at most by 276 bytes.

With this limit unmodified, the record layer fragments a 16385 byte message into two records.

For example, to send an ApplicationData record 1 byte larger than the *TLS* specified limit, use the following code:

```
node = node.add_child(SetMaxRecordSize(2**16-1))  # "unlimited"
node = node.add_child(ApplicationDataGenerator(bytearray(b'A' * 16385)))
```

You can find a usage example in: test-record-size-limit.py.

6.4 Message fragmentation

Tlsfuzzer provides methods to control fragmentation and sending of the messages.

6.4.1 Splitting messages

To send one higher level message in more than one record, you can use *split_message()*, *PopMessageFromList*, and *FlushMessageList*.

The *split_message()* requires a `list()` object to pass the created fragments to the other two nodes. It sends the first fragment at that point. *PopMessageFromList* takes one fragment from the list and sends it. *FlushMessageList* takes all remaining fragments from the list and sends them in one record. If a message has a post-send action, they execute it after sending the last fragment.

For example, to send a ClientHello in two records, the first of 2 bytes length, use the following code:

```
ciphres = [CipherSuite.TLS_RSA_WITH_AES_128_CBC_SHA,
           CipherSuite.TLS_EMPTY_RENEGOTIATION_INFO_SCSV]
msg_gen = ClientHelloGenerator(ciphers)
fragment_list = []
node = node.add_child(split_message(msg_gen, fragment_list, 2))
node = node.add_child(FlushMessageList(fragment_list))
```

You can find a usage example in: test-large-hello.py.

CHAPTER 7

Connection state

Processing of nodes like *ExpectServerHello* or *ClientKeyExchangeGenerator* updates the state of the connection: the encryption keys, handshake hashes, and so on.

To perform more complex handshakes, you need to take more direct control of some of those variables.

7.1 Opening and closing the connection

To open a *TCP*: connection use the *Connect* node. It provides also ability to control the record layer protocol version using the `version` parameter and setting the amount of time runner waits for messages from peer using the `timeout` parameter.

In contrast, the *Close* node closes the *TCP* connection and doesn't accept any parameters.

For example, to start session resumption, you need to close the old connection and open a new one.

You can find a usage example of them in: test-tls13-session-resumption.py.

7.2 Handshake hashes

TLS uses a running hash of all exchanged messages to verify the integrity of the handshake and to perform signatures in CertificateVerify messages.

Before session resumption or renegotiation, you need to zero out, or reset, those hashes.

The *tlsfuzzer.messages.ResetHandshakeHashes* node allows to do that.

For example, to start renegotiation right after finishing a handshake use the following code:

```
node = node.add_child(ExpectChangeCipherSpec())
node = node.add_child(ExpectFinished())
node = node.add_child(ResetHandshakeHashes())
node = node.add_child(ClientHelloGenerator(ciphers,
```

(continues on next page)

(continued from previous page)

```
                                                session_id=bytearray(0),
                                                extensions=ext))
```

You can find a usage example in: test-legacy-renegotiation.py.

7.3 Renegotiation info

During secure renegotiation peers send the value of last Finished message in the `renegotiation_info` extension. If you use automatic generators for processing this extension, you need to reset the values from Finished before a new handshake using *ResetRenegotiationInfo*.

For example, to start session resumption using session IDs use the following code:

```
...
node = node.add_child(ExpectClose())
node = node.add_child(Close())
node = node.add_child(Connect(host, port))
node = node.add_child(ResetHandshakeHashes())
node = node.add_child(ResetRenegotiationInfo())
node = node.add_child(ClientHelloGenerator(
    ciphers,
    extensions={ExtensionType.renegotiation_info:None}))
```

You can find a usage example in: test-sessionID-resumption.py.

7.4 Clearing encryption settings

Tlsfuzzer allows also disabling encryption for sent messages. To reset the context for sending records, use the *ResetWriteConnectionState*.

For example, to send an unencrypted Finished message use the following code:

```
...
node = node.add_child(ExpectFinished())
node = node.add_child(ResetWriteConnectionState())
node = node.add_child(FinishedGenerator())
```

You can find a usage example in test-tls13-finished-plaintext.py.

7.5 Clearing post-handshake authentication context

A client associates its reply to the server's CertificateRequest message by sending it with the same context. To pass that association around *ExpectCertificateRequest*, *CertificateGenerator*, *CertificateVerifyGenerator*, and *FinishedGenerator* accept the `context` keyword argument. If the runner executes the same conversation many times, as it does with `sanity` test cases, that context needs resetting between runs. *ClearContext* provides this functionality.

For example, to handle a single post-handshake authentication use the following code:

```
...
context = []
node = node.add_child(ExpectCertificateRequest(context=context))
node = node.add_child(CertificateGenerator(
    X509CertChain([cert]), context=context))
node = node.add_child(CertificateVerifyGenerator(
    private_key, context=context))
node = node.add_child(FinishedGenerator(context=context))
node = node.add_child(ClearContext(context))
```

You can find a usage example in test-tls13-post-handshake-auth.py.

CHAPTER 8

Statistical analysis

As cryptographic security depends on proper use of primitives, tests need to verify contents of parameters.

Tlsfuzzer allows collecting some of the parameters to perform the analysis later.

8.1 AES-GCM nonces

The *AES-GCM* construction in *TLS* 1.2 uses explicit nonces. Peers select the nonce themselves and send it to their peer.

Since reusing the nonce breaks the encryption, the peers must not do that.

To collect the nonces sent by peer, use the `CollectNonces` node. Place it right after encryption negotiation: after `ExpectChangeCipherSpec` node.

After executing the connection through runner, the passed in array has the nonces selected by the peer saved as binary strings—one for every record received.

See the test-aes-gcm-nonces.py script for example how to verify that they monotonically increase.

8.2 Saving cryptographic parameters

Unlike nonces, negotiation or advertising of other cryptographic parameters happens just once per connection. To save those parameters use the `CopyVariables` node. For full list of supported parameters see the class documentation, you can find definitions of the names in the *TLS RFC*s.

As a parameter this node accepts a dictionary in which keys specify names of parameters to collect. The node appends collected parameters to the values of the dictionary.

For example, to check the uniqueness of `random` values sent in `ServerHello`, use the following code:

```
collected_randoms = []
variables_check = {"ServerHello.random": collected_randoms}
conversation = Connect(host, port)
node = conversation
ciphers = [CipherSuite.TLS_RSA_WITH_AES_128_CBC_SHA,
           CipherSuite.TLS_EMPTY_RENEGOTIATION_INFO_SCSV]
node = node.add_child(ClientHelloGenerator(ciphers))
node = node.add_child(ExpectServerHello())
node = node.add_child(CopyVariables(variables_check))
node = node.add_child(Close())

runner = Runner(conversation)
runner.run()
runner = Runner(conversation)
runner.run()
assert collected_randoms[0] != collected_randoms[1]
```

You can use the same `variables_check` or `collected_randoms` with more than one *`CopyVariables`*, it appends new values to the arrays, it doesn't replace the arrays.

You can find a usage example of it in: test-serverhello-random.py.

Tip: Tlsfuzzer provides a simple function to verify uniqueness of parameters in such a dictionary: *`uniqueness_check()`*.

CHAPTER 9

Timing analysis

As cryptographic implementations process secret data, they need to ensure that side effects of processing that data do not reveal information about the secret data.

When an implementation takes different amounts of time to process the messages we consider it a timing side-channel. When such side-channels reflect the contents of the processed messages we call them timing oracles.

One of the oldest timing oracles is the attack described by Daniel Bleichenbacher against RSA key exchange. You can test for it using the test-bleichenbacher-timing.py script.

The other is related to de-padding and verifying MAC values in CBC ciphertexts, the newest iteration of which is called Lucky Thirteen. You can test for it using the test-lucky13.py script.

9.1 Environment setup

As the scripts measure the time it takes a server to reply to a message, a server running alone on a machine, with no interruptions from other services or processes will provide statistically significant results with fewest observations.

9.1.1 Hardware selection

You will want a server with at least 3 physical cores: one to run the OS, tlsfuzzer script, etc., one to run the tcpdump process (to ensure consistent timestamping of captured packets) and one to run the system under test (to ensure consistent response times).

While you can run the tests against a network server, this manual doesn't describe how to ensure low latency and low jitter to such system under test.

It's better to use a desktop or server system with sufficient cooling as thermal throttling is common for laptops running heavy workloads resulting in jitter and overall inconsistent results.

9.1.2 OS configuration

To ensure the lowest level of noise in measurement, configure the system to isolate cores for tcpdump and the system under test.

Red Hat Enterprise Linux 8

To isolate CPUs on RHEL-8, install the following packages:

```
dnf install -y tuned tuned-utils tuned-profiles-cpu-partitioning
```

And add the following code to `/etc/tuned/cpu-partitioning-variables.conf` file:

```
isolated_cores=2-10
no_balance_cores=2-10
```

Then apply the profile:

```
tuned-adm profile cpu-partitioning
```

and restart the system to apply the changes to the kernel.

Then you can install tlsfuzzer dependencies to speed-up the test execution:

```
dnf install python3 python3-devel tcpdump gmp-devel swig mpfr-devel \
libmpc openssl-devel make gcc gcc-c++ git libmpc-devel python3-six

pip3 install m2crypto gmpy2
pip3 install --pre tlslite-ng
```

And the general requirements to collect and analyse timing results:

```
pip install -r requirements-timing.txt
```

Note: Because the tests use packet capture to collect timing information and they buffer the messages until all of them have been created, the use of `m2crypto` and `gmpy2` does not have an effect on collected data points, using them will only make tlsfuzzer run the tests at a higher frequency.

9.2 Testing theory

Because the measurements the test performs are statistical by nature, the scripts can't just take a mean of observations and compare them with means of observations of other tests—that will not provide quantifiable results. This is caused by the fact that the measurements don't follow a simple and well-defined distribution, in many cases they are multimodal and not normal. That means that the scripts need to use statistical tests to check if the observations differ significantly or not.

Most statistical tests work in terms of hypothesis testing. Scripts use Wilcoxon signed-rank test and the Sign test to compare samples. After executing it against two sets of observations (samples), it outputs a "p-value"—a probability of getting such samples, if they were taken from the same population. A high p-value (close to 1) means that the samples likely came from the same source while a small value (close to 0, smaller than 0.05) means that it's unlikely that they came from the same source distribution.

Generally, script assumes that the p-values below 0.05 mean that the values came from different distributions, i.e. the server behaves differently for the two provided inputs.

But such small values are expected even if the samples were taken from the same distribution if the number of performed tests is large, so you need to check if those values are no more common than expected.

If the samples did indeed come from the same population, then the distribution of p-values will follow a uniform distribution with values between 0 and 1.

You can use this property to check if not only the failures (small p-values) occur not more often than expected, but to check for more general inconsistency in p-values (as higher probability of small p-values means that large p-values occur less often).

The scripts perform the Kolmogorov–Smirnov test to test the uniformity of p-values of the Wilcoxon tests and the sign test.

The test scripts allow setting the sample size as it has impact on the smallest effect size that the test can detect. Generally, with both of the used tests, the sample size must be proportional to $1/e^2$ to detect effect of size e. That is, to detect a 0.1% difference between expected values of samples, the samples must have at least 1000 observations each. The actual number depends on multiple factors (including the particular samples in question), but it's a good starting point. Also, it means that if you wish to decrease the reported confidence interval by a factor of 10, you must execute the script with 100 times as many repetitions (as 10^2=100).

Note that this effect size is proportional to magnitude of any single observation, at the same time things like size of pre master secret or size of MAC are constant, thus configuring the server to use fast ciphers and small key sizes for RSA will make the test detect smaller (absolute) effect sizes, if they exist.

Finally, the scripts take the pair of samples most dissimilar to each other and estimate the difference and the 99% confidence interval for the difference to show the estimated effect size.

You can also use the following R script to calculate the confidence intervals for the difference between a given pair of samples using the Wilcoxon test:

```
df <- read.csv('timing.csv', header=F)
data <- df[,2:length(df[1,])]
# print headers (names of tests)
df[,1]
# run Wilcoxon signed-rank test between second and third sample,
# report 99% confidence interval for the difference:
wilcox.test(as.numeric(data[2,]), as.numeric(data[3,]), paired=T, conf.int=T, conf.
↪level=0.99)
```

To put into practical terms, a run with 10000 observations, checking a server with a 100µs response time will not detect a timing side channel that's smaller than 0.01µs (40 cycles on a 4GHz CPU).

9.3 Running the tests

To run the tests:

1. Select a machine with sufficient cooling and a multi-core CPU
2. Use methods mentioned before to create isolated cores, watch out for hyperthreading
3. For RSA tests use small key (1024 bit), for CBC tests use a fast cipher and hash.
4. Start the server on one of the isolated cores, e.g.:

```
taskset --cpu-list 2,3 openssl s_server -key key.pem -cert cert.pem -www
```

5. Start the test script, provide the IDs of different isolated cores:

```
PYTHONPATH=. python3 scripts/test-lucky13.py -i lo --repeat 100 --cpu-list 4,5
```

6. Wait (a long) time
7. Inspect summary of the analysis, or move the test results to a host with newer python and analyse it there.

Note: Since both using pinned cores and collecting packets requires root permissions, execute the previously mentioned commands as root.

Warning: The tests use `tcpdump` to collect packets to a file and analyse it later. To process tests with large `--repeat` parameter, you need a machine with a large amount of disk space: at least 350MiB with 20 tests at 10000 repeats.

9.3.1 Test argument interface

Any test that collects timing information provides the following argument interface. Specifying the network interface that packet capture should listen on should be enough to time the tests.

Argument	Required	Description
`-i interface`	Yes	Interface to run tcpdump on
`-o dir`	No	Output directory (default `/tmp`)
`--repeat rep`	No	Repeat each test `rep` times (default 100)
`--cpu-list`	No	Core IDs to use for running tcpdump (default none)

9.3.2 Executing the test, extraction and analysis

Tests can be executed the same way as any non-timing tests, just make sure the current user has permissions to run tcpdump or use sudo. As an example, the Bleichenbacher test is extended to use the timing functionality:

```
sudo PYTHONPATH=. python scripts/test-bleichenbacher-timing.py -i lo
```

By default, if `dpkt` dependency is available, the extraction will run right after the timing packet capture. In case you want to run the extraction on another machine (e.g. you were not able to install the optional dependencies) you can do this by providing the log, the packet capture and server port and hostname (or ip) to the analysis script. Resulting file will be outputted to the specified folder.

```
PYTHONPATH=. python tlsfuzzer/extract.py -h localhost -p 4433 \
-c capture.pcap -l log.csv -o /tmp/results/
```

Timing runner will also launch analysis, if its dependencies are available. Again, in case you need to run it later, you can do that by providing the script with an output folder where extraction step put the `timing.csv` file.

```
PYTHONPATH=. python tlsfuzzer/analysis.py -o "/tmp/results"
```

With large sample sizes, to avoid exhausting available memory and to speed up the analysis, you can skip the generation of some graphs using the `--no-ecdf-plot`, `--no-scatter-plot` and `--no-conf-interval-plot`. That last option disables generation of the `bootstrapped_means.csv` file too.

9.3.3 External timing data

The `extract.py` can also process data collected by some external source (be it packet capture closer to server under test or an internal probe inside the server).

The provided csv file must have a header and one column. While the file can contain additional data points at the beginning, the very last data point must correspond to the last connection made by tlsfuzzer.

Place such file in the directory (in this example named `timings-log.csv`) with the `log.csv` file and execute:

```
PYTHONPATH=. python tlsfuzzer/extract.py -l /tmp/results/log.csv \
-o /tmp/results --raw-times /tmp/results/timings-log.csv
```

Warning: The above mentioned command will overrite the timings extracted from the `capture.pcap` file!

Then run `analysis.py` as in the case of data extracted from `capture.pcap` file:

```
PYTHONPATH=. python tlsfuzzer/analysis.py -o "/tmp/results"
```

9.3.4 Combining results from multiple runs

You can use the `combine.py` script to combine the results from runs.

The script checks if the set of executed probes match in all the files, but you need to ensure that the environments of the test execution match too.

To combine the runs, provide the output directory (`out-dir` here) and paths to one or more `timing.csv` files:

```
PYTHONPATH=. python tlsfuzzer/combine.py -o out-dir \
in_1596892760/timing.csv in_1596892742/timing.csv
```

Warning: The script overwrites the `timing.csv` in the output directory!

After combining the `timing.csv` files, execute analysis as usual.

Tip: `combine.py` is the only script able to read the old format of `timing.csv` files. Use it with a single input file to covert from old file format (where all results for a given probe ware listed in a single line) to the new file format (where all results for a given probe are in a single column)

9.4 Interpreting the results

You should start the inspection of test results with the `scatter_plot.png` graph. It plots all of the collected connection times. There is also a zoomed-in version that will be much more readable in case of much larger outliers. You can find it in the `scatter_plot_zoom_in.png` file. If you can see that there is a periodicity to the collected measurements, or the values can be collected in similarly looking groups, that means that the data is autocorrelated (or, in other words, not-independent) and simple summary statistics like mean, median, or quartiles are not representative of the samples.

The next set of graphs show the overall shape of the samples. The `box_plot.png` shows the 5th percentile, 1st quartile, median, 3rd quartile and 95th percentile. The `ecdf_plot.png` shows the measured (that is, empirical) cumulative distribution function. The `ecdf_plot_zoom_in.png` shows only the values between 1st and 95th percentile, useful in case of few very large outliers. The "steps" visible in the graph inform us if the distibution is unimodal (like the common normal distribution) or if it is multimodal. Multimodality is another property that makes simple summary statistics like mean or median not representative of the sample.

To compare autocorrelated samples we need to compare the differences between pairs of samples. The `diff_scatter_plot.png` shows the differences of all the samples when compared to the first sample (numbered 0). The `diff_ecdf_plot.png` is the ECDF counterpart to the scatter plot. Here, if the graph is symmetrical then the results from the Wilcoxon signed-rank test are meaningful. If the graph is asymmetric focus on sign test results. The `diff_ecdf_plot_zoom_in_98.png`, `diff_ecdf_plot_zoom_in_33.png`, and `diff_ecdf_plot_zoom_in_10.png` show just the central 98, 33, and 10 percentiles respectively of the graph (to make estimating small differences between samples easier).

Finally, the `conf_interval_plot_mean.png`, `conf_interval_plot_median.png`, `conf_interval_plot_trim_mean_05.png`, `conf_interval_plot_trim_mean_25.png`, and `conf_interval_plot_trimean.png` show the mean, median, trimmed mean (5%), trimmed mean (25%), and trimean respecively, of the differences between samples together with bootstrapped confidence interval for them. For an implementation without a timing side channel present, all the graphs should intersect with the horizonal 0 line. If a graph does not intersect with the 0 line, then the number of heights of it from the 0 line suggests how strong is the confidence in the presence of side channel on an exponential scale.

As mentioned previously, the script executes tests in three stages, first is the Wilcoxon signed-rank test and sign test between all the samples, second is the uniformity test of those results, third is the Friedman test.

Warning: The implementation of Friedman test uses an approximation using Chi-squared distribution. That means the results of it are reliable only with many samples (at least 5, optimally 10). You should ignore it for such small runs. It's also invalid in case of just two samples (used conversations).

The sign test is performed in three different ways: the default, used for determining presence of the timing side-channel, is the two-sided variant, saved in the `report.csv` file as the `Sign test`. The two other ways, the `Sign test less` and `Sign test greater` test the hypothesis that the one sample stochastically dominates the other. High p-values here aren't meangingful (i.e. you can get a p-value == 1 even if the alternative is not statistically significant even at alpha=0.05). Very low values of a `Sign test less` mean that the *second* sample is unlikely to be smaller than the *first* sample. Those tests are more sensitive than the confidence intervals for median, so you can use them to test the theory if the timing signal depends on some parameters, like the length of pre-master secret in RSA key exchange or place of the first mismatched byte in CBC MAC.

The code also calculates the dependent t-test for paired samples, but as the timings generally don't follow the normal distribution, it severly underestimates the difference between samples (it is strongly influenced by outliers). The results from it are not taken into account to decide failure of the overall timing test.

If either the KS-tests of uniformity of p-values, or the Friedman test fails, you should inspect the individual test p-values.

If one particular set of tests consistently scores low when compared to other tests (e.g. "very long (96-byte) pre master secret" and "very long (124-byte) pre master secret" from `test-bleichenbacher-timing.py`) but high when compared with each-other, that strongly points to a timing side-channel in the system under test.

If the timing signal has a high relative magnitude (one set of tests slower than another set by 10%), then you can also use the generated `box_plot.png` graph to see it. For small differences with large sample sizes, the differences will be statistically detectable, even if not obvious from from the box plot. You can use the `conf_interval_plot*.png` graphs to see the difference between samples and the first sample together with the 95% confidence interval for them.

The script prints the numerical value for confidence interval for mean, median, trimmed mean (with 5% of observervations on either end ignored), trimmed mean (with 25% of smalles and biggest observations ignored), and trimean of differences of the pair of two most dissimilar probes. It also writes them to the `report.txt` file.

The `report.csv` file includes the exact p-values for the statistical tests executed as well as the calculated descriptive statistics of distribution of differences: the mean, standard deviation (SD), median, interquartile range (IQR, as well as the median absolute deviation (MAD). Note that the mean and SD are very sensitive to outliers, the other three measures are more robust. The calculated MAD already includes the conversion factor so for a normal distribution it can be compared directly to SD.

The `sample_stats.csv` file include the calculated mean, median, and MAD for the samples themselves (i.e. not the differences between samples). You can use this data to estimate the smallest detectable difference between samples for a given sample size.

Using R you can also manually generate `conf_interval_plot_mean.png` graph, but note that this will take about an hour for 21 tests and samples with 1 million observations each on a 4 core/8 thread 2GHz CPU:

```
library(tidyr)
library(ggplot2)
library(dplyr)
library(data.table)
library(boot)
df <- fread('timing.csv', header=F)
data <- data.frame(t(df[,2:length(df[1,])]))
colnames(data) <- as.matrix(df[,1:10])[,1]
df <- 0
R = 5000
rsq <- function(data, indices) {
  d <- data[indices]
  return(mean(d, trim=0.25))
}
data2 = replicate(R, 0)
data2 = cbind(data2)
date()
for (i in c(2:length(data[1,]))) {
  a = boot(data[,1]-data[,i], rsq, R=R, parallel="multicore",
           simple=TRUE, ncpus=8)
  data2 = cbind(data2, a$t)
}
date()
data2 = data.frame(data2)
data2 %>% gather(key="MeasureType", value="Delay") %>%
ggplot( aes(x=factor(MeasureType, level=colnames(data2)), y=Delay,
            fill=factor(MeasureType, level=colnames(data2)))) +
geom_violin() + xlab("Test ID") +
ylab("Trimmed mean of differences [s]") + labs(fill="Test ID")
colnames(data)
```

9.5 Writing new test scripts

The `TimingRunner` repeatedly runs tests with `tcpdump` capturing packets in the background. The timing information is then extracted from that `tcpdump` capture, only the response time to the last client message is extracted from the capture.

9.5.1 Test structure

After processing these arguments, one would proceed to write the test as usual, probably adding a `sanity` test case and tests cases relating to the feature under test. The example script `test-conversation.py` can be used as a starting point.

After it is clear, that all the tests passed, timing of the tests can be executed. Please note that any tests with `sanity` prefix will be ignored in the timing run. Start by importing the `TimingRunner` class. Because the timing information collection adds some extra dependencies, it is necessary to wrap everything related to timing in an if statement:

```
if TimingRunner.check_tcpdump():
```

Now, the `TimingRunner` class can be initialized with the name of the currently run test, list of conversations (`sampled_tests` in the reference scripts), output directory (the `-o` argument), TLS server host and port, and finally the network interface from the `-i` argument.

Next step is to generate log with random order of test cases for each run. This is done by calling the function `generate_log()` from the `TimingRunner` instance. This function takes the familiar `run_only` and `run_exclude` variables that can filter what tests should be run. Note that this function will exclude any tests named "sanity". The last argument to this function is how many times each test should be run (`--repeat` argument). The log is saved in the output directory.

The last step is to call `run()` function from the `TiminingRunner` instance in order to launch tcpdump and begin iterating over the tests. Provided you were able to install the timing dependencies, this will also launch extraction that will process the packet capture, and output the timing information associated with the test class into a csv file, and analysis that will generate a report with statistical test results and supporting plots.

CHAPTER 10

Integrating in CI

While you can write one-off test cases using tlsfuzzer to test a specific issue, tlsfuzzer caters to *CI* environments.

10.1 Preparation

Configuration of the server has significant impact on its behaviour. For example, you can't test *ECDSA* cipher suites without an *ECDSA* certificate available for the server.

To verify all features of a server implementation you need to enable all of them in the server. In case the features conflict with each-other, you need to test them separately by running the relevant tests with every configuration.

As a full-featured implementation has a lot of independent parameters, you can find yourself in a situation where you don't have enough computer resources to test all combinations of parameters. In such case you may want to apply combinatorial testing, or pairwise testing, to keep the required amount of server configurations manageable.

10.1.1 Configuration variables

You should take into consideration the following aspects of server configuration:

1. How many and what types of certificates the server has set up
    - especially relevant for differences between `rsaEncryption` and `rsassa-pss` in Subject Public Key Info
2. Test with *SNI* enabled and disabled
3. Testing the default host vs *SNI* host of a *SNI*-enabled server
4. Test with different security levels enabled in the library
    - for OpenSSL: cipherstring with `@SECLEVEL=3` vs `@SECLEVEL=0`
    - for GnuTLS: priority string with `NORMAL` vs `SECURE256:%PROFILE_HIGH`
5. No client certificates, requesting client certificates, or requiring client certificates

6. Test with *ALPN* (or *NPN*) enabled and disabled
7. Session tickets enabled or not
    - For example in OpenSSL it influences the *kind* of tickets issued by OpenSSL in *TLS* 1.3
8. Test with support for 0-RTT enabled and disabled
    - for example, even with 0-RTT disabled, the server still must process 0-RTT ClientHello but ignore the early data
9. Enabled protocol versions (e.g. with *TLS* 1.2 implemented but only *TLS* 1.1 enabled, the server must not abort connection starting with a *TLS* 1.1 ClientHello with `TLS_FALLBACK_SCSV`
10. Changes to configuration before session establishment and resumption (e.g. resuming a session with a now disabled cipher; or processing 0-RTT data with its cipher disabled)
11. Integrated with different applications—callbacks can change important parts of implementation behaviour
12. Large client or server certificates
13. The private key residing in an *HSM* or a smart card
    - different features supported in the smart card—things like RSA-PSS or SHA-384
14. Testing with specific extension or feature of the protocol disabled, enabled, or required
15. Server running under valgrind, compiled with ubsan, asan, etc.
16. Force-enabled features deprecated in later protocol versions (e.g. PKCS#1 v1.5 SHA-1 signatures enabled through configuration should not enable them in *TLS* 1.3)
17. Server running on different hardware (different assembly implementations in use, AES-NI support, SSE3, etc.)
18. Interactions between implementation versions and session resumptions— test what happens when a client resumes a session from old library with new server (and vice versa, to simulate server downgrade)

10.2 Running tests

Since the included tests expect strict adherence to *RFC*s, you can expect that executing them for the first time will find a lot of issues. As such, you should start with running them one by one, manually, inspecting test results and checking if they pass.

Many tests verify behaviour unrelated to main feature under test, indicated by the name and the summary printed at the end of execution. Such tests provide extra command line options to make them more aligned with behaviour of the tested implementation.

Warning: Some tests allow changing expected alert description for the negative tests. Before introducing any such modifcations you should have a good understanding of *TLS* and oracle attacks—you need to verify that similarly malformed messages result in the exact same alerts. Please note that some features (e.g. padding in *CBC* mode ciphers) have more than one script that tests them, so you need to adjust invocations of all relevant scripts.

If you find differences between script-expected behaviour and actually observed behaviour of the system under test, inspect the source code to determine the root cause of the issue. Commonly the implementation detects the wrong behaviour of the peer but returns a wrong alert. While technically those are compliance issues and you should fix them (see also the section *describing reasons for strict alert description checking*), they don't cause interoperability or security issues, so you can postpone fixing them.

While working on a test script, you should adjust its parameters so that it matches the server configuration. If a script expects different behaviour, you can either disable running the failing test case by specifying its name as a parameter to `-e` option or mark it as an "expected failure" by specifying its name as a parameter to `-x` option. In the latter case, you can also specify a substring to match the printed error against with the `-X` option. Using `-x` ensures that resolving the bug causes the test suite to "notice" the new behaviour. Pairing it with `-X` option ensures that the *way* the test fails doesn't change.

Scripts by default require less than 10 seconds to execute against a local server (using a mid-range CPU from 2020). You can use the `-n` option to control how many tests to execute in a script. To execute all tests in a script, specify `-n 0`.

10.3 Automation scripts

Tlsfuzzer ships with a few scripts to make using it in *CI* easier: `verify-scripts-json.py`, `scripts_retention.py`, and `verify-multiple-jsons.py`.

The `scripts_retention.py` one starts servers based on passed configuration file. To ensure reliable execution, it verifies that the server can accept connections before running the test cases. The script uses `server_hostname` and `server_port` to verify readiness of the server to accept connections.

Tip: The `server_command` can specify the server to run or a command necessary to reconfigure the server. In latter case, it needs to run for as long as `scripts_retention.py` executes test scripts for— `scripts_retention.py` aborts testing when this command exits.

A test case marked with `"exp_pass": false` needs to fail, otherwise the script counts it as a failure.

You can find example configuration files in `tests` directory: tlslite-ng.json and tlslite-ng-random-subset.json. The latter one is part of the *CI* for tlsfuzzer.

The `verify-multiple-jsons.py` and `verify-scripts-json.py` check if specified json files reference all tests in the `scripts` directory. You should use them when migrating to new version of tlsfuzzer to verify that you don't skip any newly added scripts.

CHAPTER 11

Writing test coverage for RFCs

As *TLS* has long history and offers support for multiple algorithms, features often interact with each-other. When planning test coverage for a new feature or a new extension, you need to create test cases for those possible interactions.

11.1 The standard

To create a good test coverage you need to have detailed standard—one that includes expected behaviour both for expected and unexpected behaviour from the peer.

See text below for detailed list of possible interactions.

To ensure that the standard is detailed enough you should get involved with it at the draft stage and check if it includes things like expected alert descriptions or iteractions with session resumption. *IETF* doesn't allow changes to already published *RFC*s.

11.2 Planning test coverage

You should read the standard and annotate it in places where it prescribes specific behaviour. Then turn those annotations into test scripts or test cases.

The list of cases to consider for inclusion in a script:

1. **sanity check**: simple configuration to check if connection with server works and that server continues to work after all tests are finished
2. **specific alert**: when testing for error conditions, the ExpectAlert expects one specific value for the alert (with one exception, instead of `handshake_failure` server may send `insufficient_security`); test needs to fail if the server doesn't send an alert (with the exception of `close_notify` alert)
3. **renegotiation**: check how feature interacts with renegotiation, do renegotiation handshakes need to include it, can it use different settings in it, can the client omit it? (only for *TLS* 1.2 and earlier)

4. **resumption**: check how feature interacts with session resumption, do clients have to advertise it in the resumed client hello, does the server need to advertise it in resumed session, can it use different settings in resumed session? Can clients drop it in resumed session?
5. **client certificates**: does the feature relates to handling certificates, does the client need to send it too when the server asked for it in the extensions of CertificateRequest (*TLS* 1.3)?
6. **virtual hosts**: does the standard permin for different behaviour for different virtual hosts, either defined by different *SNI*/`server_name` or by *ALPN*? If not explicitly allowed, do you test for consistent behaviour?
7. **undefined codepoints**: does the standard describe behaviour with undefined code points (see for example at `signature_algorithms` extension), does the peer has to ignore them? What happens if the connection has only undefined (essentially unknown to the peer) points?
    - use the undefined code points first then place well known in the list—after all if new types are added, they should be more secure than the old types—to verify that peer doesn't have hardcoded limits for list lengths
8. **disabled codepoints**: do the disabled codepoints not cause issues when they are advertised together with the good codepoints (e.g. MD5 hashes in `signature_algorithms`, Koblitz curves in `supported_groups` in *TLS* 1.3)?
9. **duplicated codepoints**: does the standard allow for duplicated entries (items with the same values)? If not, does the peer reject them? What happens if script sends a lot of duplicated, known, but unsupported (or disabled) entries before sending something that the server accepts? (this checks if peer does not abort parsing after filling a short list of known values)

    Note: unless a definition for a particular list doesn't prohibit duplicated values (like for `key_shares` or extensions as a whole), *TLS does* allow for duplicated values

    ---

10. **invalid combinations**: check if peer doesn't accept different codepoints in place of a correct one, like a RSA signature with a RSA certificate but advertised as an ECDSA signature, or a `ecdsa_secp521r1_sha512` signature with a secp256r1 certificate in *TLS* 1.3 (to verify that the peer checks the whole value and doesn't short-circuit some checks)
11. **large lists**: check if the server can process a list that has max size but is otherwise well-formed (check if server doesn't have inherent limits for processing)
12. **empty values**: many arrays in *TLS* have min length greater than zero, check if peer rejects empty values in such cases
13. *PRF* **interaction**: for features that depend on master secret calculation, do they work as expected with ciphers that use "protocol default *PRF*" (*TLS* 1.1 ciphers in *TLS* 1.2), SHA-256, or SHA-384 as *PRF*?
14. **padded/truncated lengths**: do you check if values like extension payloads or array elements are not accepted when they have less data than expected or more data than they should (i.e. mismatch between different length fields)
15. **padded/truncated data**: for fields like signatures or finished values, the data needs to be of very specific size, check if it is padded or truncated (either left or right, both for padding and truncation, or completely omitted, length included), it is rejected
16. **impossible lengths**: for lists of same sized items, some sizes are impossible, like odd lengths for ClientHello cipher list or `signature_algorithms` list of schemes, check if peer rejects this kind of values (including one-byte payload)
17. **HelloRetryRequest interaction**: for extensions sent in ClientHello that affect *TLS* 1.3 sessions, verify if server detects a modified version of it in 2nd ClientHello and aborts the connection
    - also check if server detects adding of it to 2nd ClientHello or dropping of it from 2nd CH and aborts the connection

18. **TLS 1.3 padding**: if the extension affects handling of records, how does it interact with TLS 1.3 record layer padding? do the size limits apply to padding or not?
19. **0-RTT**: does it impact handling of `early_data` messages?
20. **version confusion**: does the peer reject values or messages valid in one version of protocol when test uses them in another? (e.g. it needs to reject `rsa_pkcs1_sha224` signatures in *TLS* 1.3 and KeyUpdate messages in *TLS* 1.2)
21. **documentation**: does the script describe (in *printed* messages) what is the general purpose of it?
22. **version**: does the script report its version? (you should make it a monotonically increasing value, updated with every change to the test scenarios)
23. **protocol version/protocol type**: does protocol version of *TLS* have an impact? is it applicable to DTLS? (tlsfuzzer doesn't support DTLS, yet: #55)
24. **interaction with other extensions**: does the test need to test the scenario also with other extensions?
    1. `extended_master_secret`: deos the scenario interact with derived secrets of keys?
    2. `encrypt_then_mac` (EtM): does the scenario interact with record layer? record sizes? ciphers?
25. **renegotiation and resumption**: how does the extension behave when the renegotiation *and* resumption is combined, especially when the resumed session had the status of extension different than the session in which the renegotiation happens? See also points 3. and 4. (no support for such test cases, see #591)
26. **invalid extension for message**: RFC 8446 Section 4.2 states that peers must reject recognised extensions in unexpected messages (like `cookie` in CertificateRequest) with `illegal_parameter`. Verify that peer behaves in this way.

CHAPTER 12

Projects using tlsfuzzer

Besides the internal use of tlsfuzzer in Red Hat few projects adopted it in upstream testing.

12.1 GnuTLS

You can find the most complete integration of tlsfuzzer in the GnuTLS project. The configuration files for it reside in the main source repository, in tests/suite/tls-fuzzer directory.

To see it in action compile GnuTLS as usual and go to the `tests/suite` directory. There execute `make check TESTS=tls-fuzzer/tls-fuzzer-nocert-tls13.sh`.

12.2 NSS

The other project with few test scripts automated is the NSS library. You can find the test scripts in the nss/tests/tlsfuzzer directory.

CHAPTER 13

Glossary

AEAD Authenticated Encryption with Associated Data, a mode of operation for symmetric ciphers that processes messages and optional additional data as atomic objects: the decryption provides data only if integrity of data is verified, encryption provides ciphertext only when all the data was provided to the encryption function.

AES Advanced Encryption Standard is a symmetric block cipher.

AES-CCM *AEAD* mode of Advanced Encryption Standard (*AES*) that combines counter mode with the CBC-MAC algorithm. In *TLS* those ciphers require version 1.2 or 1.3.

AES-CCM8 *AES-CCM* with 8 byte long authentication tag.

AES-GCM Advanced Encryption Standard in Galois Counter Mode is an *AEAD* cipher, it encrypts and authenticates data with one operation. In *TLS* those ciphers require version 1.2 or 1.3.

ALPN Application Layer Protocol Negotiation is a *TLS* extension allowing for co-existence of multiple applications protocols on the same *TCP* or UDP port. Commonly used to negotiate HTTP/2 over HTTP/1.1.

CBC Cipher Block Chaining, an encryption mode for block ciphers, used since SSLv2 until TLS 1.2.

CI Continuous Integration is a development practice in which changes are merged to `master` branch, commonly after the test coverage for the project is executed.

CMAC Cipher-based *MAC*

ECDHE Implementation of Diffie-Hellman key exchange algorithm over elliptic curves.

ECDSA Elliptic Curve Digital Signature Algorithm uses the Digital Signature Algorithm with elliptic curves instead of finite field groups. It's an asymmetric cryptosystem, similar to RSA.

GMAC Galois *MAC*, commonly used as part of the *AES-GCM* cipher.

HMAC Hash-based *MAC*, commonly used with CBC mode ciphers in *TLS* before version 1.3

HSM Hardware Security Module is usually an extension card that is tasked with secure storage of private keys. Some HSMs also provide hardware acceleration for cryptographic operations.

IETF Internet Engineering Task Force is an organisation responsible for providing specifications of protocols used over the Internet.

IV Initialisation Vector, a value used to influence the generated ciphertext, unlike the key, it doesn't have to remain secret

MAC Message Authentication Code is the generic name for data used to verify integrity of the received data. This data is called an authentication tag. There are many MACs defined: *HMAC*, *CMAC*, or *GMAC*.

NPN Next Protocol Negotiation is a *TLS* extension allowing for use of multiple application layer protocols on the same port. Not standardised. Obsoleted by *ALPN*.

PKIX Public Key Infrastructure for the Internet, described use of X.509 certificates in Internet protocols.

PRF Pseudo-Random Function is used to sanitise random values to prepare them for use as keys in encryption. *TLS* 1.0 and 1.1 uses combination of MD5 and SHA1. *TLS* 1.2 and 1.3 use SHA-256 or SHA-384 based algorithms depending on cipher suite negotiated.

RFC Request For Comments are standards published by Internet Engineering Task Force, an open standards organisation.

RSA Rivest Shamir Adleman is an asymmetric cryptosystem commonly used for signing messages or encrypting keys.

SNI Server Name Indication, also known as `server_name`, is a *TLS* extension for negotiatiating connections to "Virtual Hosts". It allows a server to distinguish requests for different hostnames sharing a single IP address.

SSL Secure Sockets Layer is an old cryptographic network protocol. It has orginated in Netscape in the early 1990's. Currently replaced by *TLS*.

SUT System Under Test is the device or implementation that the tests are verifying. Excludes tlsfuzzer itself or systems necessary to execute it or tlsfuzzer.

TCP Transport Control Protocol is a stream protocol that provides reliable delivery over the Internet Protocol.

TLS Transport Layer Security is a cryptographic network protocol defined in a series of *RFC* documents, newest of which is RFC8446.

CHAPTER 14

tlsfuzzer API

14.1 tlsfuzzer package

Library with tests and fuzzers for the TLS protocol.

Use objects in *tlsfuzzer.messages* to create objects that will be sent to the other side of a SSL or TLS connection and *tlsfuzzer.expect* to process messages received from the other side. The *tlsfuzzer.runner* will execute those prepared messages.

Objects that have direct effect on the state of encryption of the connection: *ExpectChangeCipherSpec*, *ExpectServerHello*, *ChangeCipherSpecGenerator*, *ExpectFinished* and *FinishedGenerator*.

14.1.1 Subpackages

tlsfuzzer.utils package

Various convenience functions, unrelated to TLS or crypto

Submodules

tlsfuzzer.utils.lists module

Utility functions for lists.

tlsfuzzer.utils.lists.**natural_sort_keys**(*key*, *_nsre=re.compile('([0-9]+)')*)

Split the key into a sortable list for the `sorted()` builtin.

Natural sort sorts words using dictionary order and numbers using numerical order, so `ab20` will be placed before `ab100`.

Used with `sorted()` like this:

```
a = dict()
b = sorted(a, key=natural_sort_keys)
```

Parameters **key** – key used for sorting

Return type list

tlsfuzzer.utils.ordered_dict module

Compatibility wrapper of OrderedDict.

14.1.2 Submodules

tlsfuzzer.analysis module

tlsfuzzer.expect module

Parsing and processing of received TLS messages

class tlsfuzzer.expect.**Expect**(*content_type*)

Bases: *tlsfuzzer.tree.TreeNode*

Base class for objects handling message readers

_repr(*attributes*)

Return a text representation of the object.

Parameters **attributes** (*list(str)*) – names of attributes of the object that will be included in the text representation

add_child(*child*)

Sets the parameter as the child of the node

Returns the child node

get_all_siblings()

Return iterator with all siblings of node

Return type iterator

is_command()

Flag to tell that the object is a message processor

is_expect()

Flag to tell if the object is a message processor

is_generator()

Flag to tell that the object is not a message generator

is_match(*msg*)

Checks if the object can handle message

Note that the msg is a raw, unparsed message of indicated type that requires calling write() to get a raw bytearray() representation of it

Parameters **msg** (*tlslite.messages.Message*) – raw message to check

process(*state*, *msg*)

Process the message and update the state accordingly.

Parameters

- **state** (tlsfuzzer.runner.ConnectionState) – current connection state, needs to be updated after parsing the message by inheriting classes
- **msg** (*tlslite.messages.Message*) – raw message to parse

class tlsfuzzer.expect.**ExpectAlert**(*level=None*, *description=None*)

Bases: *tlsfuzzer.expect.Expect*

Processing TLS Alert message

_repr(*attributes*)

Return a text representation of the object.

Parameters **attributes** (*list(str)*) – names of attributes of the object that will be included in the text representation

add_child(*child*)

Sets the parameter as the child of the node

Returns the child node

get_all_siblings()

Return iterator with all siblings of node

Return type iterator

is_command()

Flag to tell that the object is a message processor

is_expect()

Flag to tell if the object is a message processor

is_generator()

Flag to tell that the object is not a message generator

is_match(*msg*)

Checks if the object can handle message

Note that the msg is a raw, unparsed message of indicated type that requires calling write() to get a raw bytearray() representation of it

Parameters **msg** (*tlslite.messages.Message*) – raw message to check

process(*state*, *msg*)

Process the message and update the state accordingly.

Parameters

- **state** (tlsfuzzer.runner.ConnectionState) – current connection state, needs to be updated after parsing the message by inheriting classes
- **msg** (*tlslite.messages.Message*) – raw message to parse

class tlsfuzzer.expect.**ExpectApplicationData**(*data=None*, *size=None*, *output=None*)

Bases: *tlsfuzzer.expect.Expect*

Processing Application Data message

_repr(*attributes*)

Return a text representation of the object.

Parameters **attributes** (*list(str)*) – names of attributes of the object that will be included in the text representation

add_child(*child*)
Sets the parameter as the child of the node

Returns the child node

get_all_siblings()
Return iterator with all siblings of node

Return type iterator

is_command()
Flag to tell that the object is a message processor

is_expect()
Flag to tell if the object is a message processor

is_generator()
Flag to tell that the object is not a message generator

is_match(*msg*)
Checks if the object can handle message

Note that the msg is a raw, unparsed message of indicated type that requires calling write() to get a raw bytearray() representation of it

Parameters **msg** (*tlslite.messages.Message*) – raw message to check

process(*state*, *msg*)
Process the message and update the state accordingly.

Parameters

- **state** (tlsfuzzer.runner.ConnectionState) – current connection state, needs to be updated after parsing the message by inheriting classes
- **msg** (*tlslite.messages.Message*) – raw message to parse

class tlsfuzzer.expect.**ExpectCertificate**(*cert_type=0*)
Bases: *tlsfuzzer.expect.ExpectHandshake*

Processing TLS Handshake protocol Certificate messages

static _cmp_eq(*our*, *recv*, *field_type=None*, *f_str=None*)
Check if expected value matched received, if defined.

If our is not None, compare with recv. If they don't match, try translating them with field_type.toStr() method and rise AssertionError with message formatted with f_str. First parameter to .format() will be expected value and the second one will be the received one

static _cmp_eq_list(*our*, *recv*, *field_type=None*, *f_str=None*)
Check if expected list of values matched received, if defined.

If our is not None, compare with recv. If they don't match, try translating items in the lists with field_type.toStr() method and rise AssertionError with message formatted with f_str. First parameter to .format() will be list of expected values and the second one will be the received one

classmethod _cmp_eq_or_in(*our*, *recv*, *field_type=None*, *f_str=None*)
Check if received value equals expected or is in expected list.

If our is a list or set, check if recv is in it. If our is not None, check if it's equal to recv. If they don't match or are not part of a set, try translating them with field_type.toStr() method and raise AssertionError formatted with f_str. First parameter to .format() will be the expected value and the second one witll be the received one.

_repr(*attributes*)
Return a text representation of the object.

Parameters **attributes** (*list(str)*) – names of attributes of the object that will be included in the text representation

add_child(*child*)
Sets the parameter as the child of the node

Returns the child node

get_all_siblings()
Return iterator with all siblings of node

Return type iterator

is_command()
Flag to tell that the object is a message processor

is_expect()
Flag to tell if the object is a message processor

is_generator()
Flag to tell that the object is not a message generator

is_match(*msg*)
Check if message is a given type of handshake protocol message

process(*state*, *msg*)

class tlsfuzzer.expect.**ExpectCertificateRequest**(*sig_algs=None*, *cert_types=None*, *sanity_check_cert_types=True*, *extensions=None*, *context=None*)
Bases: *tlsfuzzer.expect._ExpectExtensionsMessage*

Processing TLS Handshake protocol Certificate Request message.

static _cmp_eq(*our*, *recv*, *field_type=None*, *f_str=None*)
Check if expected value matched received, if defined.

If our is not None, compare with recv. If they don't match, try translating them with field_type.toStr() method and rise AssertionError with message formatted with f_str. First parameter to .format() will be expected value and the second one will be the received one

static _cmp_eq_list(*our*, *recv*, *field_type=None*, *f_str=None*)
Check if expected list of values matched received, if defined.

If our is not None, compare with recv. If they don't match, try translating items in the lists with field_type.toStr() method and rise AssertionError with message formatted with f_str. First parameter to .format() will be list of expected values and the second one will be the received one

classmethod _cmp_eq_or_in(*our*, *recv*, *field_type=None*, *f_str=None*)
Check if received value equals expected or is in expected list.

If our is a list or set, check if recv is in it. If our is not None, check if it's equal to recv. If they don't match or are not part of a set, try translating them with field_type.toStr() method and raise AssertionError formatted with f_str. First parameter to .format() will be the expected value and the second one witll be the received one.

_compare_extensions(*message*)
Verify that server provided extensions match exactly expected list.

static _get_autohandler(*ext_id*)

_process_extensions(*state*, *msg*)

_repr(*attributes*)
Return a text representation of the object.

Parameters **attributes** (*list(str)*) – names of attributes of the object that will be included in the text representation

static _sanity_check_cert_types(*cert_request*)
Verify that the CertificateRequest is self-consistent.

add_child(*child*)
Sets the parameter as the child of the node

Returns the child node

get_all_siblings()
Return iterator with all siblings of node

Return type iterator

is_command()
Flag to tell that the object is a message processor

is_expect()
Flag to tell if the object is a message processor

is_generator()
Flag to tell that the object is not a message generator

is_match(*msg*)
Check if message is a given type of handshake protocol message

process(*state*, *msg*)
Check received Certificate Request

class tlsfuzzer.expect.**ExpectCertificateStatus**
Bases: *tlsfuzzer.expect.ExpectHandshake*

Processing of CertificateStatus message from RFC 6066.

static _cmp_eq(*our*, *recv*, *field_type=None*, *f_str=None*)
Check if expected value matched received, if defined.

If our is not None, compare with recv. If they don't match, try translating them with field_type.toStr() method and rise AssertionError with message formatted with f_str. First parameter to .format() will be expected value and the second one will be the received one

static _cmp_eq_list(*our*, *recv*, *field_type=None*, *f_str=None*)
Check if expected list of values matched received, if defined.

If our is not None, compare with recv. If they don't match, try translating items in the lists with field_type.toStr() method and rise AssertionError with message formatted with f_str. First parameter to .format() will be list of expected values and the second one will be the received one

classmethod _cmp_eq_or_in(*our*, *recv*, *field_type=None*, *f_str=None*)
Check if received value equals expected or is in expected list.

If our is a list or set, check if recv is in it. If our is not None, check if it's equal to recv. If they don't match or are not part of a set, try translating them with field_type.toStr() method and raise AssertionError formatted with f_str. First parameter to .format() will be the expected value and the second one witll be the received one.

_repr(*attributes*)
Return a text representation of the object.

Parameters **attributes** (*list(str)*) – names of attributes of the object that will be included in the text representation

add_child(*child*)
Sets the parameter as the child of the node

Returns the child node

get_all_siblings()
Return iterator with all siblings of node

Return type iterator

is_command()
Flag to tell that the object is a message processor

is_expect()
Flag to tell if the object is a message processor

is_generator()
Flag to tell that the object is not a message generator

is_match(*msg*)
Check if message is a given type of handshake protocol message

process(*state*, *msg*)
Process the message and update the state accordingly.

Parameters

- **state** (tlsfuzzer.runner.ConnectionState) – current connection state, needs to be updated after parsing the message by inheriting classes
- **msg** (*tlslite.messages.Message*) – raw message to parse

class tlsfuzzer.expect.**ExpectCertificateVerify**(*version=None*, *sig_alg=None*)
Bases: *tlsfuzzer.expect.ExpectHandshake*

Processing TLS Handshake protocol Certificate Verify messages.

static _cmp_eq(*our*, *recv*, *field_type=None*, *f_str=None*)
Check if expected value matched received, if defined.

If our is not None, compare with recv. If they don't match, try translating them with field_type.toStr() method and rise AssertionError with message formatted with f_str. First parameter to .format() will be expected value and the second one will be the received one

static _cmp_eq_list(*our*, *recv*, *field_type=None*, *f_str=None*)
Check if expected list of values matched received, if defined.

If our is not None, compare with recv. If they don't match, try translating items in the lists with field_type.toStr() method and rise AssertionError with message formatted with f_str. First parameter to .format() will be list of expected values and the second one will be the received one

classmethod _cmp_eq_or_in(*our*, *recv*, *field_type=None*, *f_str=None*)
Check if received value equals expected or is in expected list.

If our is a list or set, check if recv is in it. If our is not None, check if it's equal to recv. If they don't match or are not part of a set, try translating them with field_type.toStr() method and raise AssertionError formatted with f_str. First parameter to .format() will be the expected value and the second one witll be the received one.

_repr(*attributes*)
Return a text representation of the object.

Parameters **attributes** (*list(str)*) – names of attributes of the object that will be included in the text representation

add_child(*child*)
Sets the parameter as the child of the node

Returns the child node

get_all_siblings()
Return iterator with all siblings of node

Return type iterator

is_command()
Flag to tell that the object is a message processor

is_expect()
Flag to tell if the object is a message processor

is_generator()
Flag to tell that the object is not a message generator

is_match(*msg*)
Check if message is a given type of handshake protocol message

process(*state*, *msg*)

class tlsfuzzer.expect.**ExpectChangeCipherSpec**
Bases: *tlsfuzzer.expect.Expect*

Processing TLS Change Cipher Spec messages.

Note: In SSLv3 up to TLS 1.2, the message modifies the state of record layer to expect encrypted records *after* receiving this message. In case of renegotiation, record layer will expect records encrypted with the newly negotiated keys. In TLS 1.3 it has no effect on record layer encryption.

_repr(*attributes*)
Return a text representation of the object.

Parameters **attributes** (*list(str)*) – names of attributes of the object that will be included in the text representation

add_child(*child*)
Sets the parameter as the child of the node

Returns the child node

get_all_siblings()
Return iterator with all siblings of node

Return type iterator

is_command()
Flag to tell that the object is a message processor

is_expect()
Flag to tell if the object is a message processor

is_generator()
Flag to tell that the object is not a message generator

is_match(*msg*)
Checks if the object can handle message

Note that the msg is a raw, unparsed message of indicated type that requires calling write() to get a raw bytearray() representation of it

Parameters **msg** (*tlslite.messages.Message*) – raw message to check

process(*state*, *msg*)

class tlsfuzzer.expect.**ExpectClose**
Bases: *tlsfuzzer.expect.Expect*

Virtual message signifying closing of TCP connection

_repr(*attributes*)
Return a text representation of the object.

Parameters **attributes** (*list(str)*) – names of attributes of the object that will be included in the text representation

add_child(*child*)
Sets the parameter as the child of the node

Returns the child node

get_all_siblings()
Return iterator with all siblings of node

Return type iterator

is_command()
Flag to tell that the object is a message processor

is_expect()
Flag to tell if the object is a message processor

is_generator()
Flag to tell that the object is not a message generator

is_match(*msg*)
Checks if the object can handle message

Note that the msg is a raw, unparsed message of indicated type that requires calling write() to get a raw bytearray() representation of it

Parameters **msg** (*tlslite.messages.Message*) – raw message to check

process(*state*, *msg*)
Close our side

class tlsfuzzer.expect.**ExpectEncryptedExtensions**(*extensions=None*)
Bases: *tlsfuzzer.expect._ExpectExtensionsMessage*

Processing of the TLS handshake protocol Encrypted Extensions message

static _cmp_eq(*our*, *recv*, *field_type=None*, *f_str=None*)
Check if expected value matched received, if defined.

If our is not None, compare with recv. If they don't match, try translating them with field_type.toStr() method and rise AssertionError with message formatted with f_str. First parameter to .format() will be expected value and the second one will be the received one

static _cmp_eq_list(*our*, *recv*, *field_type=None*, *f_str=None*)
Check if expected list of values matched received, if defined.

If our is not None, compare with recv. If they don't match, try translating items in the lists with field_type.toStr() method and rise AssertionError with message formatted with f_str. First parameter to .format() will be list of expected values and the second one will be the received one

classmethod _cmp_eq_or_in(*our*, *recv*, *field_type=None*, *f_str=None*)
Check if received value equals expected or is in expected list.

If our is a list or set, check if recv is in it. If our is not None, check if it's equal to recv. If they don't match or are not part of a set, try translating them with field_type.toStr() method and raise AssertionError formatted with f_str. First parameter to .format() will be the expected value and the second one witll be the received one.

_compare_extensions(*message*)
Verify that server provided extensions match exactly expected list.

_compare_extensions_in_ee(*srv_exts*, *cln_hello*)
Verify that server provided extensions match exactly expected list.

static _get_autohandler(*ext_id*)

_process_extensions(*state*, *srv_exts*)
Check if extensions are correct.

_repr(*attributes*)
Return a text representation of the object.

Parameters **attributes** (*list(str)*) – names of attributes of the object that will be included in the text representation

add_child(*child*)
Sets the parameter as the child of the node

Returns the child node

get_all_siblings()
Return iterator with all siblings of node

Return type iterator

is_command()
Flag to tell that the object is a message processor

is_expect()
Flag to tell if the object is a message processor

is_generator()
Flag to tell that the object is not a message generator

is_match(*msg*)
Check if message is a given type of handshake protocol message

process(*state*, *msg*)
Process the message and update the state accordingly.

Parameters

- **state** (tlsfuzzer.runner.ConnectionState) – current connection state, needs to be updated after parsing the message by inheriting classes
- **msg** (*tlslite.messages.Message*) – raw message to parse

class tlsfuzzer.expect.**ExpectFinished**(*version=None*)
Bases: *tlsfuzzer.expect.ExpectHandshake*

Processing TLS handshake protocol Finished message.

Note: In TLS 1.3 the message will modify record layer to start *sending* records with encryption using the `client_handshake_traffic_secret` keys. It will also modify the record layer to start expecting the records to be encrypted with `server_application_traffic_secret` keys.

---

static _cmp_eq(*our*, *recv*, *field_type=None*, *f_str=None*)
Check if expected value matched received, if defined.

If our is not None, compare with recv. If they don't match, try translating them with field_type.toStr() method and rise AssertionError with message formatted with f_str. First parameter to .format() will be expected value and the second one will be the received one

static _cmp_eq_list(*our*, *recv*, *field_type=None*, *f_str=None*)
Check if expected list of values matched received, if defined.

If our is not None, compare with recv. If they don't match, try translating items in the lists with field_type.toStr() method and rise AssertionError with message formatted with f_str. First parameter to .format() will be list of expected values and the second one will be the received one

classmethod _cmp_eq_or_in(*our*, *recv*, *field_type=None*, *f_str=None*)
Check if received value equals expected or is in expected list.

If our is a list or set, check if recv is in it. If our is not None, check if it's equal to recv. If they don't match or are not part of a set, try translating them with field_type.toStr() method and raise AssertionError formatted with f_str. First parameter to .format() will be the expected value and the second one witll be the received one.

_repr(*attributes*)
Return a text representation of the object.

Parameters **attributes** (*list(str)*) – names of attributes of the object that will be included in the text representation

add_child(*child*)
Sets the parameter as the child of the node

Returns the child node

get_all_siblings()
Return iterator with all siblings of node

Return type iterator

is_command()
Flag to tell that the object is a message processor

is_expect()
Flag to tell if the object is a message processor

is_generator()
Flag to tell that the object is not a message generator

is_match(*msg*)
Check if message is a given type of handshake protocol message

process(*state*, *msg*)

class tlsfuzzer.expect.**ExpectHandshake**(*content_type*, *handshake_type*)
Bases: *tlsfuzzer.expect.ExpectMessage*

Common methods for handling TLS Handshake protocol messages

static _cmp_eq(*our*, *recv*, *field_type=None*, *f_str=None*)
Check if expected value matched received, if defined.

If our is not None, compare with recv. If they don't match, try translating them with field_type.toStr() method and rise AssertionError with message formatted with f_str. First parameter to .format() will be expected value and the second one will be the received one

static _cmp_eq_list(*our*, *recv*, *field_type=None*, *f_str=None*)
Check if expected list of values matched received, if defined.

If our is not None, compare with recv. If they don't match, try translating items in the lists with field_type.toStr() method and rise AssertionError with message formatted with f_str. First parameter to .format() will be list of expected values and the second one will be the received one

classmethod _cmp_eq_or_in(*our*, *recv*, *field_type=None*, *f_str=None*)
Check if received value equals expected or is in expected list.

If our is a list or set, check if recv is in it. If our is not None, check if it's equal to recv. If they don't match or are not part of a set, try translating them with field_type.toStr() method and raise AssertionError formatted with f_str. First parameter to .format() will be the expected value and the second one witll be the received one.

_repr(*attributes*)
Return a text representation of the object.

Parameters **attributes** (*list(str)*) – names of attributes of the object that will be included in the text representation

add_child(*child*)
Sets the parameter as the child of the node

Returns the child node

get_all_siblings()
Return iterator with all siblings of node

Return type iterator

is_command()
Flag to tell that the object is a message processor

is_expect()
Flag to tell if the object is a message processor

is_generator()
Flag to tell that the object is not a message generator

is_match(*msg*)
Check if message is a given type of handshake protocol message

process(*state*, *msg*)
Process the message and update the state accordingly.

Parameters

- **state** (tlsfuzzer.runner.ConnectionState) – current connection state, needs to be updated after parsing the message by inheriting classes
- **msg** (*tlslite.messages.Message*) – raw message to parse

class tlsfuzzer.expect.**ExpectHeartbeat**(*message_type=2*, *payload=None*, *padding_size=None*)

Bases: *tlsfuzzer.expect.ExpectMessage*

Processing of heartbeat messages.

static _cmp_eq(*our*, *recv*, *field_type=None*, *f_str=None*)

Check if expected value matched received, if defined.

If our is not None, compare with recv. If they don't match, try translating them with field_type.toStr() method and rise AssertionError with message formatted with f_str. First parameter to .format() will be expected value and the second one will be the received one

static _cmp_eq_list(*our*, *recv*, *field_type=None*, *f_str=None*)

Check if expected list of values matched received, if defined.

If our is not None, compare with recv. If they don't match, try translating items in the lists with field_type.toStr() method and rise AssertionError with message formatted with f_str. First parameter to .format() will be list of expected values and the second one will be the received one

classmethod _cmp_eq_or_in(*our*, *recv*, *field_type=None*, *f_str=None*)

Check if received value equals expected or is in expected list.

If our is a list or set, check if recv is in it. If our is not None, check if it's equal to recv. If they don't match or are not part of a set, try translating them with field_type.toStr() method and raise AssertionError formatted with f_str. First parameter to .format() will be the expected value and the second one witll be the received one.

_repr(*attributes*)

Return a text representation of the object.

Parameters **attributes** (*list(str)*) – names of attributes of the object that will be included in the text representation

add_child(*child*)

Sets the parameter as the child of the node

Returns the child node

get_all_siblings()

Return iterator with all siblings of node

Return type iterator

is_command()

Flag to tell that the object is a message processor

is_expect()

Flag to tell if the object is a message processor

is_generator()

Flag to tell that the object is not a message generator

is_match(*msg*)

Checks if the object can handle message

Note that the msg is a raw, unparsed message of indicated type that requires calling write() to get a raw bytearray() representation of it

Parameters **msg** (*tlslite.messages.Message*) – raw message to check

process(*state*, *msg*)
Check if the msg meets the requirements for the message.

class tlsfuzzer.expect.**ExpectHelloRequest**(*description=None*)
Bases: *tlsfuzzer.expect.ExpectHandshake*

Processing of TLS handshake protocol hello request message.

static _cmp_eq(*our*, *recv*, *field_type=None*, *f_str=None*)
Check if expected value matched received, if defined.

If our is not None, compare with recv. If they don't match, try translating them with field_type.toStr() method and rise AssertionError with message formatted with f_str. First parameter to .format() will be expected value and the second one will be the received one

static _cmp_eq_list(*our*, *recv*, *field_type=None*, *f_str=None*)
Check if expected list of values matched received, if defined.

If our is not None, compare with recv. If they don't match, try translating items in the lists with field_type.toStr() method and rise AssertionError with message formatted with f_str. First parameter to .format() will be list of expected values and the second one will be the received one

classmethod _cmp_eq_or_in(*our*, *recv*, *field_type=None*, *f_str=None*)
Check if received value equals expected or is in expected list.

If our is a list or set, check if recv is in it. If our is not None, check if it's equal to recv. If they don't match or are not part of a set, try translating them with field_type.toStr() method and raise AssertionError formatted with f_str. First parameter to .format() will be the expected value and the second one witll be the received one.

_repr(*attributes*)
Return a text representation of the object.

Parameters **attributes** (*list(str)*) – names of attributes of the object that will be included in the text representation

add_child(*child*)
Sets the parameter as the child of the node

Returns the child node

get_all_siblings()
Return iterator with all siblings of node

Return type iterator

is_command()
Flag to tell that the object is a message processor

is_expect()
Flag to tell if the object is a message processor

is_generator()
Flag to tell that the object is not a message generator

is_match(*msg*)
Check if message is a given type of handshake protocol message

process(*state*, *msg*)
Parse, verify and process the message.

class tlsfuzzer.expect.**ExpectHelloRetryRequest**(*extensions=None*, *version=None*, *cipher=None*)

Bases: *tlsfuzzer.expect.ExpectServerHello*

Processing of the TLS 1.3 HelloRetryRequest message.

static _check_against_hrr(*state*, *srv_hello*)

_check_downgrade_protection(*srv_hello*)
Verify that server provided downgrade protection as specified in RFC 8446, Section 4.1.3

static _cmp_eq(*our*, *recv*, *field_type=None*, *f_str=None*)
Check if expected value matched received, if defined.

If our is not None, compare with recv. If they don't match, try translating them with field_type.toStr() method and rise AssertionError with message formatted with f_str. First parameter to .format() will be expected value and the second one will be the received one

static _cmp_eq_list(*our*, *recv*, *field_type=None*, *f_str=None*)
Check if expected list of values matched received, if defined.

If our is not None, compare with recv. If they don't match, try translating items in the lists with field_type.toStr() method and rise AssertionError with message formatted with f_str. First parameter to .format() will be list of expected values and the second one will be the received one

classmethod _cmp_eq_or_in(*our*, *recv*, *field_type=None*, *f_str=None*)
Check if received value equals expected or is in expected list.

If our is a list or set, check if recv is in it. If our is not None, check if it's equal to recv. If they don't match or are not part of a set, try translating them with field_type.toStr() method and raise AssertionError formatted with f_str. First parameter to .format() will be the expected value and the second one witll be the received one.

_compare_extensions(*message*)
Verify that server provided extensions match exactly expected list.

static _extract_version(*msg*)
Extract the real version from the message if TLS 1.3 is in use.

static _get_autohandler(*ext_id*)

_process_extensions(*state*, *cln_hello*, *srv_hello*)
Check if extensions are correct.

_repr(*attributes*)
Return a text representation of the object.

Parameters **attributes** (*list(str)*) – names of attributes of the object that will be included in the text representation

_setup_tls13_handshake_keys(*state*)
Prepare handshake ciphers for the HRR handling

add_child(*child*)
Sets the parameter as the child of the node

Returns the child node

get_all_siblings()
Return iterator with all siblings of node

Return type iterator

is_command()
Flag to tell that the object is a message processor

is_expect()
Flag to tell if the object is a message processor

is_generator()
Flag to tell that the object is not a message generator

is_match(*msg*)
Check if message is a given type of handshake protocol message

process(*state*, *msg*)
Process the message and update state accordingly

Parameters

- **state** (ConnectionState) – overall state of TLS connection
- **msg** (*Message*) – TLS Message read from socket

class tlsfuzzer.expect.**ExpectKeyUpdate**(*message_type=None*)
Bases: *tlsfuzzer.expect.ExpectHandshake*

Processing of post-handshake KeyUpdate message from RFC 8446

static _cmp_eq(*our*, *recv*, *field_type=None*, *f_str=None*)
Check if expected value matched received, if defined.

If our is not None, compare with recv. If they don't match, try translating them with field_type.toStr() method and rise AssertionError with message formatted with f_str. First parameter to .format() will be expected value and the second one will be the received one

static _cmp_eq_list(*our*, *recv*, *field_type=None*, *f_str=None*)
Check if expected list of values matched received, if defined.

If our is not None, compare with recv. If they don't match, try translating items in the lists with field_type.toStr() method and rise AssertionError with message formatted with f_str. First parameter to .format() will be list of expected values and the second one will be the received one

classmethod _cmp_eq_or_in(*our*, *recv*, *field_type=None*, *f_str=None*)
Check if received value equals expected or is in expected list.

If our is a list or set, check if recv is in it. If our is not None, check if it's equal to recv. If they don't match or are not part of a set, try translating them with field_type.toStr() method and raise AssertionError formatted with f_str. First parameter to .format() will be the expected value and the second one witll be the received one.

_repr(*attributes*)
Return a text representation of the object.

Parameters **attributes** (*list(str)*) – names of attributes of the object that will be included in the text representation

add_child(*child*)
Sets the parameter as the child of the node

Returns the child node

get_all_siblings()
Return iterator with all siblings of node

Return type iterator

is_command()
Flag to tell that the object is a message processor

is_expect()
Flag to tell if the object is a message processor

is_generator()
Flag to tell that the object is not a message generator

is_match(*msg*)
Check if message is a given type of handshake protocol message

process(*state*, *msg*)
Parse, verify and process the message.

class tlsfuzzer.expect.**ExpectMessage**(*content_type*)
Bases: *tlsfuzzer.expect.Expect*

Common methods for handling TLS messages.

static _cmp_eq(*our*, *recv*, *field_type=None*, *f_str=None*)
Check if expected value matched received, if defined.

If our is not None, compare with recv. If they don't match, try translating them with field_type.toStr() method and rise AssertionError with message formatted with f_str. First parameter to .format() will be expected value and the second one will be the received one

static _cmp_eq_list(*our*, *recv*, *field_type=None*, *f_str=None*)
Check if expected list of values matched received, if defined.

If our is not None, compare with recv. If they don't match, try translating items in the lists with field_type.toStr() method and rise AssertionError with message formatted with f_str. First parameter to .format() will be list of expected values and the second one will be the received one

classmethod _cmp_eq_or_in(*our*, *recv*, *field_type=None*, *f_str=None*)
Check if received value equals expected or is in expected list.

If our is a list or set, check if recv is in it. If our is not None, check if it's equal to recv. If they don't match or are not part of a set, try translating them with field_type.toStr() method and raise AssertionError formatted with f_str. First parameter to .format() will be the expected value and the second one witll be the received one.

_repr(*attributes*)
Return a text representation of the object.

Parameters **attributes** (*list(str)*) – names of attributes of the object that will be included in the text representation

add_child(*child*)
Sets the parameter as the child of the node

Returns the child node

get_all_siblings()
Return iterator with all siblings of node

Return type iterator

is_command()
Flag to tell that the object is a message processor

is_expect()
Flag to tell if the object is a message processor

is_generator()
Flag to tell that the object is not a message generator

is_match(*msg*)
Checks if the object can handle message

Note that the msg is a raw, unparsed message of indicated type that requires calling write() to get a raw bytearray() representation of it

Parameters **msg** (*tlslite.messages.Message*) – raw message to check

process(*state*, *msg*)
Process the message and update the state accordingly.

Parameters

- **state** (tlsfuzzer.runner.ConnectionState) – current connection state, needs to be updated after parsing the message by inheriting classes
- **msg** (*tlslite.messages.Message*) – raw message to parse

class tlsfuzzer.expect.**ExpectNewSessionTicket**(*description=None*)
Bases: *tlsfuzzer.expect.ExpectHandshake*

Processing TLS handshake protocol new session ticket message.

static _cmp_eq(*our*, *recv*, *field_type=None*, *f_str=None*)
Check if expected value matched received, if defined.

If our is not None, compare with recv. If they don't match, try translating them with field_type.toStr() method and rise AssertionError with message formatted with f_str. First parameter to .format() will be expected value and the second one will be the received one

static _cmp_eq_list(*our*, *recv*, *field_type=None*, *f_str=None*)
Check if expected list of values matched received, if defined.

If our is not None, compare with recv. If they don't match, try translating items in the lists with field_type.toStr() method and rise AssertionError with message formatted with f_str. First parameter to .format() will be list of expected values and the second one will be the received one

classmethod _cmp_eq_or_in(*our*, *recv*, *field_type=None*, *f_str=None*)
Check if received value equals expected or is in expected list.

If our is a list or set, check if recv is in it. If our is not None, check if it's equal to recv. If they don't match or are not part of a set, try translating them with field_type.toStr() method and raise AssertionError formatted with f_str. First parameter to .format() will be the expected value and the second one witll be the received one.

_repr(*attributes*)
Return a text representation of the object.

Parameters **attributes** (*list(str)*) – names of attributes of the object that will be included in the text representation

add_child(*child*)
Sets the parameter as the child of the node

Returns the child node

get_all_siblings()
Return iterator with all siblings of node

Return type iterator

is_command()
Flag to tell that the object is a message processor

is_expect()
Flag to tell if the object is a message processor

is_generator()
Flag to tell that the object is not a message generator

is_match(*msg*)
Check if message is a given type of handshake protocol message

process(*state*, *msg*)
Parse, verify and process the message.

class tlsfuzzer.expect.**ExpectNoMessage**(*timeout=0.1*)
Bases: *tlsfuzzer.expect.Expect*

Virtual message signifying timeout on message listen.

Variables **timeout** (*int or float*) – how long to wait for message before giving up, in seconds, can be float

_repr(*attributes*)
Return a text representation of the object.

Parameters **attributes** (*list(str)*) – names of attributes of the object that will be included in the text representation

add_child(*child*)
Sets the parameter as the child of the node

Returns the child node

get_all_siblings()
Return iterator with all siblings of node

Return type iterator

is_command()
Flag to tell that the object is a message processor

is_expect()
Flag to tell if the object is a message processor

is_generator()
Flag to tell that the object is not a message generator

is_match(*msg*)
Checks if the object can handle message

Note that the msg is a raw, unparsed message of indicated type that requires calling write() to get a raw bytearray() representation of it

Parameters **msg** (*tlslite.messages.Message*) – raw message to check

process(*state*, *msg*)
Do nothing.

class tlsfuzzer.expect.**ExpectSSL2Alert**(*error=None*)
Bases: *tlsfuzzer.expect.ExpectHandshake*

Processing of SSLv2 Handshake protocol alert messages

static _cmp_eq(*our*, *recv*, *field_type=None*, *f_str=None*)
Check if expected value matched received, if defined.

If our is not None, compare with recv. If they don't match, try translating them with field_type.toStr() method and rise AssertionError with message formatted with f_str. First parameter to .format() will be expected value and the second one will be the received one

static _cmp_eq_list(*our*, *recv*, *field_type=None*, *f_str=None*)
Check if expected list of values matched received, if defined.

If our is not None, compare with recv. If they don't match, try translating items in the lists with field_type.toStr() method and rise AssertionError with message formatted with f_str. First parameter to .format() will be list of expected values and the second one will be the received one

classmethod _cmp_eq_or_in(*our*, *recv*, *field_type=None*, *f_str=None*)
Check if received value equals expected or is in expected list.

If our is a list or set, check if recv is in it. If our is not None, check if it's equal to recv. If they don't match or are not part of a set, try translating them with field_type.toStr() method and raise AssertionError formatted with f_str. First parameter to .format() will be the expected value and the second one witll be the received one.

_repr(*attributes*)
Return a text representation of the object.

Parameters **attributes** (*list(str)*) – names of attributes of the object that will be included in the text representation

add_child(*child*)
Sets the parameter as the child of the node

Returns the child node

get_all_siblings()
Return iterator with all siblings of node

Return type iterator

is_command()
Flag to tell that the object is a message processor

is_expect()
Flag to tell if the object is a message processor

is_generator()
Flag to tell that the object is not a message generator

is_match(*msg*)
Check if message is a given type of handshake protocol message

process(*state*, *msg*)
Analyse the error message

class tlsfuzzer.expect.**ExpectServerHello**(*extensions=None*, *version=None*, *resume=False*, *cipher=None*, *server_max_protocol=None*, *description=None*)
Bases: *tlsfuzzer.expect._ExpectExtensionsMessage*

Parsing TLS Handshake protocol Server Hello messages.

Processing of the ServerHello message updates the record layer to the version advertisied by the server. Use *SetRecordVersion* to change it earlier to send records with different versions.

Note: Receiving of the ServerHello in TLS 1.3 influences record layer encryption. After the message is received, the `client_handshake_traffic_secret` and `server_handshake_traffic_secret` is derived and record layer is configured to expect encrypted records on the *receiving* side.

Variables **description** (*str*) – identifier to print when processing of the node fails

static _check_against_hrr(*state*, *srv_hello*)

_check_downgrade_protection(*srv_hello*)
Verify that server provided downgrade protection as specified in RFC 8446, Section 4.1.3

static _cmp_eq(*our*, *recv*, *field_type=None*, *f_str=None*)
Check if expected value matched received, if defined.

If our is not None, compare with recv. If they don't match, try translating them with field_type.toStr() method and rise AssertionError with message formatted with f_str. First parameter to .format() will be expected value and the second one will be the received one

static _cmp_eq_list(*our*, *recv*, *field_type=None*, *f_str=None*)
Check if expected list of values matched received, if defined.

If our is not None, compare with recv. If they don't match, try translating items in the lists with field_type.toStr() method and rise AssertionError with message formatted with f_str. First parameter to .format() will be list of expected values and the second one will be the received one

classmethod _cmp_eq_or_in(*our*, *recv*, *field_type=None*, *f_str=None*)
Check if received value equals expected or is in expected list.

If our is a list or set, check if recv is in it. If our is not None, check if it's equal to recv. If they don't match or are not part of a set, try translating them with field_type.toStr() method and raise AssertionError formatted with f_str. First parameter to .format() will be the expected value and the second one witll be the received one.

_compare_extensions(*message*)
Verify that server provided extensions match exactly expected list.

static _extract_version(*msg*)
Extract the real version from the message if TLS 1.3 is in use.

static _get_autohandler(*ext_id*)

_process_extensions(*state*, *cln_hello*, *srv_hello*)
Check if extensions are correct.

_repr(*attributes*)
Return a text representation of the object.

Parameters **attributes** (*list(str)*) – names of attributes of the object that will be included in the text representation

_setup_tls13_handshake_keys(*state*)
Set up the encryption keys for the TLS 1.3 handshake.

add_child(*child*)
Sets the parameter as the child of the node

Returns the child node

get_all_siblings()
Return iterator with all siblings of node

Return type iterator

is_command()
Flag to tell that the object is a message processor

is_expect()
Flag to tell if the object is a message processor

is_generator()
Flag to tell that the object is not a message generator

is_match(*msg*)
Check if message is a given type of handshake protocol message

process(*state*, *msg*)
Process the message and update state accordingly

Parameters

- **state** (ConnectionState) – overall state of TLS connection
- **msg** (*Message*) – TLS Message read from socket

class tlsfuzzer.expect.**ExpectServerHello2**(*version=None*)
Bases: *tlsfuzzer.expect.ExpectHandshake*

Processing of SSLv2 Handshake Protocol SERVER-HELLO message

static _cmp_eq(*our*, *recv*, *field_type=None*, *f_str=None*)
Check if expected value matched received, if defined.

If our is not None, compare with recv. If they don't match, try translating them with field_type.toStr() method and rise AssertionError with message formatted with f_str. First parameter to .format() will be expected value and the second one will be the received one

static _cmp_eq_list(*our*, *recv*, *field_type=None*, *f_str=None*)
Check if expected list of values matched received, if defined.

If our is not None, compare with recv. If they don't match, try translating items in the lists with field_type.toStr() method and rise AssertionError with message formatted with f_str. First parameter to .format() will be list of expected values and the second one will be the received one

classmethod _cmp_eq_or_in(*our*, *recv*, *field_type=None*, *f_str=None*)
Check if received value equals expected or is in expected list.

If our is a list or set, check if recv is in it. If our is not None, check if it's equal to recv. If they don't match or are not part of a set, try translating them with field_type.toStr() method and raise AssertionError formatted with f_str. First parameter to .format() will be the expected value and the second one witll be the received one.

_repr(*attributes*)
Return a text representation of the object.

Parameters **attributes** (*list(str)*) – names of attributes of the object that will be included in the text representation

add_child(*child*)
Sets the parameter as the child of the node

Returns the child node

get_all_siblings()
Return iterator with all siblings of node

Return type iterator

is_command()
Flag to tell that the object is a message processor

is_expect()
Flag to tell if the object is a message processor

is_generator()
Flag to tell that the object is not a message generator

is_match(*msg*)
Check if message is a given type of handshake protocol message

process(*state*, *msg*)
Process the message and update state accordingly

Parameters

- **state** (*~ConnectionState*) – overall state of TLS connection
- **msg** (*Message*) – TLS Message read from socket

class tlsfuzzer.expect.**ExpectServerHelloDone**
Bases: *tlsfuzzer.expect.ExpectHandshake*

Processing TLS Handshake protocol ServerHelloDone messages

static _cmp_eq(*our*, *recv*, *field_type=None*, *f_str=None*)
Check if expected value matched received, if defined.

If our is not None, compare with recv. If they don't match, try translating them with field_type.toStr() method and rise AssertionError with message formatted with f_str. First parameter to .format() will be expected value and the second one will be the received one

static _cmp_eq_list(*our*, *recv*, *field_type=None*, *f_str=None*)
Check if expected list of values matched received, if defined.

If our is not None, compare with recv. If they don't match, try translating items in the lists with field_type.toStr() method and rise AssertionError with message formatted with f_str. First parameter to .format() will be list of expected values and the second one will be the received one

classmethod _cmp_eq_or_in(*our*, *recv*, *field_type=None*, *f_str=None*)
Check if received value equals expected or is in expected list.

If our is a list or set, check if recv is in it. If our is not None, check if it's equal to recv. If they don't match or are not part of a set, try translating them with field_type.toStr() method and raise AssertionError formatted with f_str. First parameter to .format() will be the expected value and the second one witll be the received one.

_repr(*attributes*)
Return a text representation of the object.

Parameters **attributes** (*list(str)*) – names of attributes of the object that will be included in the text representation

add_child(*child*)
Sets the parameter as the child of the node

Returns the child node

get_all_siblings()
Return iterator with all siblings of node

Return type iterator

is_command()
Flag to tell that the object is a message processor

is_expect()
Flag to tell if the object is a message processor

is_generator()
Flag to tell that the object is not a message generator

is_match(*msg*)
Check if message is a given type of handshake protocol message

process(*state*, *msg*)

class tlsfuzzer.expect.**ExpectServerKeyExchange**(*version=None*, *cipher_suite=None*, *valid_sig_algs=None*, *valid_groups=None*, *valid_params=None*)

Bases: *tlsfuzzer.expect.ExpectHandshake*

Processing TLS Handshake protocol Server Key Exchange message

_checkParams(*server_key_exchange*)

static _cmp_eq(*our*, *recv*, *field_type=None*, *f_str=None*)
Check if expected value matched received, if defined.

If our is not None, compare with recv. If they don't match, try translating them with field_type.toStr() method and rise AssertionError with message formatted with f_str. First parameter to .format() will be expected value and the second one will be the received one

static _cmp_eq_list(*our*, *recv*, *field_type=None*, *f_str=None*)
Check if expected list of values matched received, if defined.

If our is not None, compare with recv. If they don't match, try translating items in the lists with field_type.toStr() method and rise AssertionError with message formatted with f_str. First parameter to .format() will be list of expected values and the second one will be the received one

classmethod _cmp_eq_or_in(*our*, *recv*, *field_type=None*, *f_str=None*)
Check if received value equals expected or is in expected list.

If our is a list or set, check if recv is in it. If our is not None, check if it's equal to recv. If they don't match or are not part of a set, try translating them with field_type.toStr() method and raise AssertionError formatted with f_str. First parameter to .format() will be the expected value and the second one witll be the received one.

_repr(*attributes*)
Return a text representation of the object.

Parameters **attributes** (*list(str)*) – names of attributes of the object that will be included in the text representation

add_child(*child*)
Sets the parameter as the child of the node

Returns the child node

get_all_siblings()
Return iterator with all siblings of node

Return type iterator

is_command()
Flag to tell that the object is a message processor

is_expect()
Flag to tell if the object is a message processor

is_generator()
Flag to tell that the object is not a message generator

is_match(*msg*)
Check if message is a given type of handshake protocol message

process(*state*, *msg*)
Process the Server Key Exchange message

class tlsfuzzer.expect.**ExpectVerify**
Bases: *tlsfuzzer.expect.ExpectHandshake*

Processing of SSLv2 SERVER-VERIFY message

static _cmp_eq(*our*, *recv*, *field_type=None*, *f_str=None*)
Check if expected value matched received, if defined.

If our is not None, compare with recv. If they don't match, try translating them with field_type.toStr() method and rise AssertionError with message formatted with f_str. First parameter to .format() will be expected value and the second one will be the received one

static _cmp_eq_list(*our*, *recv*, *field_type=None*, *f_str=None*)
Check if expected list of values matched received, if defined.

If our is not None, compare with recv. If they don't match, try translating items in the lists with field_type.toStr() method and rise AssertionError with message formatted with f_str. First parameter to .format() will be list of expected values and the second one will be the received one

classmethod _cmp_eq_or_in(*our*, *recv*, *field_type=None*, *f_str=None*)
Check if received value equals expected or is in expected list.

If our is a list or set, check if recv is in it. If our is not None, check if it's equal to recv. If they don't match or are not part of a set, try translating them with field_type.toStr() method and raise AssertionError formatted with f_str. First parameter to .format() will be the expected value and the second one witll be the received one.

_repr(*attributes*)
Return a text representation of the object.

Parameters **attributes** (*list(str)*) – names of attributes of the object that will be included in the text representation

add_child(*child*)
Sets the parameter as the child of the node

Returns the child node

get_all_siblings()
Return iterator with all siblings of node

Return type iterator

is_command()
Flag to tell that the object is a message processor

is_expect()
Flag to tell if the object is a message processor

is_generator()
Flag to tell that the object is not a message generator

is_match(*msg*)
Check if message is a given type of handshake protocol message

process(*state*, *msg*)
Check if the VERIFY message has expected value

class tlsfuzzer.expect.**_ExpectExtensionsMessage**(*content_type*, *msg_type*, *extensions*)
Bases: *tlsfuzzer.expect.ExpectHandshake*

Common methods of messages that have a list of extensions.

Used in ServerHello, EncryptedExtensions and CertificateRequest (in TLS 1.3)

static _cmp_eq(*our*, *recv*, *field_type=None*, *f_str=None*)
Check if expected value matched received, if defined.

If our is not None, compare with recv. If they don't match, try translating them with field_type.toStr() method and rise AssertionError with message formatted with f_str. First parameter to .format() will be expected value and the second one will be the received one

static _cmp_eq_list(*our*, *recv*, *field_type=None*, *f_str=None*)
Check if expected list of values matched received, if defined.

If our is not None, compare with recv. If they don't match, try translating items in the lists with field_type.toStr() method and rise AssertionError with message formatted with f_str. First parameter to .format() will be list of expected values and the second one will be the received one

classmethod _cmp_eq_or_in(*our*, *recv*, *field_type=None*, *f_str=None*)
Check if received value equals expected or is in expected list.

If our is a list or set, check if recv is in it. If our is not None, check if it's equal to recv. If they don't match or are not part of a set, try translating them with field_type.toStr() method and raise AssertionError formatted with f_str. First parameter to .format() will be the expected value and the second one witll be the received one.

_compare_extensions(*message*)
Verify that server provided extensions match exactly expected list.

_repr(*attributes*)
Return a text representation of the object.

Parameters **attributes** (*list(str)*) – names of attributes of the object that will be included in the text representation

add_child(*child*)
Sets the parameter as the child of the node

Returns the child node

get_all_siblings()
Return iterator with all siblings of node

Return type iterator

is_command()
Flag to tell that the object is a message processor

is_expect()
Flag to tell if the object is a message processor

is_generator()
Flag to tell that the object is not a message generator

is_match(*msg*)
Check if message is a given type of handshake protocol message

process(*state*, *msg*)
Process the message and update the state accordingly.

Parameters

- **state** (tlsfuzzer.runner.ConnectionState) – current connection state, needs to be updated after parsing the message by inheriting classes
- **msg** (*tlslite.messages.Message*) – raw message to parse

tlsfuzzer.expect.**_srv_ext_handler_psk**(*state*, *extension*, *psk_configs*)
Process the pre_shared_key extension from server.

Since it needs the psk_configurations, it can't do it automatically so it shouldn't be part of _srv_ext_handler.

tlsfuzzer.expect.**_srv_ext_handler_record_limit**(*state*, *extension*, *size=None*)
Process record_size_limit extension from server.

tlsfuzzer.expect.**clnt_ext_handler_sig_algs**(*state*, *extension*)
Check signature_algorithms or signature_algorithms_cert extension.

To be used in ClientHello and CertificateRequest.

tlsfuzzer.expect.**clnt_ext_handler_status_request**(*state*, *extension*)
Check status_request extension from initiating side.

To be used in ClientHello and CertificateRequest

tlsfuzzer.expect.**gen_srv_ext_handler_psk**(*psk_configs=()*)
Creates a handler for pre_shared_key extension from the server.

tlsfuzzer.expect.**gen_srv_ext_handler_record_limit**(*size=None*)
Create a handler for record_size_limit_extension from the server.

Note that if the extension is actually negotiated, it will override any *~SetMaxRecordSize()* before EncryptedExtensions in TLS 1.3 and before ChangeCipherSpec in TLS 1.2 and earlier.

Parameters **size** (*int*) – expected value from server, None for any valid

tlsfuzzer.expect.**hrr_ext_handler_cookie**(*state*, *extension*)
Process the cookie extension in HRR message.

tlsfuzzer.expect.**hrr_ext_handler_key_share**(*state*, *extension*)
Process the key_share extension in HRR message.

tlsfuzzer.expect.**srv_ext_handler_alpn**(*state*, *extension*)
Process the ALPN extension from server.

tlsfuzzer.expect.**srv_ext_handler_ec_point**(*state*, *extension*)
Process the ec_point_formats extension from server.

tlsfuzzer.expect.**srv_ext_handler_ems**(*state*, *extension*)
Process Extended Master Secret extension from server.

tlsfuzzer.expect.**srv_ext_handler_etm**(*state*, *extension*)
Process Encrypt then MAC extension from server.

tlsfuzzer.expect.**srv_ext_handler_heartbeat**(*state*, *extension*)
Process the heartbeat extension from server.

tlsfuzzer.expect.**srv_ext_handler_key_share**(*state*, *extension*)
Process the key_share extension from server.

tlsfuzzer.expect.**srv_ext_handler_npn**(*state*, *extension*)
Process the NPN extension from server.

tlsfuzzer.expect.**srv_ext_handler_renego**(*state*, *extension*)
Process the renegotiation_info from server.

tlsfuzzer.expect.**srv_ext_handler_sni**(*state*, *extension*)
Process the server_name extension from server.

tlsfuzzer.expect.**srv_ext_handler_status_request**(*state*, *extension*)
Process the status_request extension from server.

TLS 1.2 ServerHello specific, in TLS 1.3 the extension resides in Certificate message.

tlsfuzzer.expect.**srv_ext_handler_supp_groups**(*state*, *extension*)
Process the supported_groups from server.

tlsfuzzer.expect.**srv_ext_handler_supp_vers**(*state*, *extension*)
Process the supported_versions from server.

tlsfuzzer.extract module

tlsfuzzer.fuzzers module

Classes used for generating random (fuzzing) data

class tlsfuzzer.fuzzers.**StructuredRandom**(*vals*, *rng=None*)
Bases: object

Random data with structure.

This class allows easy creation of random data that is structured, either by having a random bytes of specific length, or intermediate bytes that are constant.

vals is a list of tuples, the first element in the tuple specifies the length of the run and the second specifies the values of the bytes in the run. If the value is None, it means the bytes should be random.

thus a `vals = [(16, 0)]` will create a bytestring of length 16, with all bytes equal to zero and `vals = [(4, None), (5, 6)]` will create a bytestring that has 4 random bytes followed by 5 bytes of value 0x06.

data
Generate the random string based on description in vals.

tlsfuzzer.fuzzers.**_normalise_groups**(*groups*, *sum_len*, *step*)
Make sure the sum of all lengths in groups is a multiple of step.

tlsfuzzer.fuzzers.**_pick_length**(*rng*, *group_min*, *group_max*)
Pick lengths of byte runs.

tlsfuzzer.fuzzers.**_pick_run_type**(*rng*, *length*)
Pick the payload of the runs with specified size.

tlsfuzzer.fuzzers.**structured_random_iter**(*count=100*, *min_length=1*, *max_length=65536*, *step=1*)
Iterator that returns a random StructuredRandom object.

Useful as a payload for TLS message plaintext

tlsfuzzer.handshake_helpers module

Methods for dealing with TLS Handshake protocol

tlsfuzzer.handshake_helpers.**calc_pending_states**(*state*)
Calculate state for pending encryption values in TLS socket

tlsfuzzer.handshake_helpers.**curve_name_to_hash_tls13**(*curve_name*)
Find the matching hash given the curve name, as specified in TLS 1.3.

tlsfuzzer.handshake_helpers.**kex_for_group**(*group*, *version=(3, 4)*)
Get a KeyExchange object for a given group and protocol version.

tlsfuzzer.helpers module

Helper functions for test scripts.

tlsfuzzer.helpers.**sig_algs_to_ids**(*names*)
Convert a string with signature algorithm names to list of IDs.

Parameters **names** (*str*) – whitespace separated list of names of hash algorithm names. Names can be specified as the legacy (TLS1.2) hash algorithm and hash type pairs (e.g. `sha256+rsa`), as a pair of numbers (e.g `4+1`) or as the new TLS 1.3 signature scheme (e.g. `rsa_pkcs1_sha256`). Full parameter string then can look like: `sha256+rsa 5+rsa rsa_pss_pss_sha256`.

Raises **AttributeError** – when the specified identifier is not defined in HashAlgorithm, SignatureAlgorithm or SignatureScheme

Returns list of tuples

tlsfuzzer.helpers.**key_share_gen**(*group*, *version=(3, 4)*)
Create a random key share for a group of a given id.

Parameters

- **group** (*int*) – TLS numerical ID from GroupName identifying the group
- **version** (*tuple*) – TLS protocol version as a tuple, as encoded on the wire

Return type *tlslite.extensions.KeyShareEntry*

tlsfuzzer.helpers.**psk_ext_gen**(*psk_settings*)
Create a PreSharedKeyExtension from given settings.

Takes a list of 2 or 3-element tuples, where the first element is an identity name, the second is the shared secret and the third is the name of the associated hash (`sha256` or `sha384`, with `sha256` being the default). The names and shared secrets need to be bytes-like objects.

Parameters **psk_settings** (*list*) – list of tuples

Returns extension

tlsfuzzer.helpers.**psk_ext_updater**(*psk_settings=()*)
Uses the provided settings to update the PSK binders in CH PSK extension.

Generator that can be used to generate the callback for the ClientHelloGenerator.modifiers setting.

See *psk_ext_gen()* for a specification of `psk_settings`.

This updater requires that the PSK extension be the last one in ClientHello.

Please note that if the ClientHello is subsequently modified (either by modifiers placed after this one or generic message fuzzers) after this updater was run, the binders it has created will likely become invalid. This is because

the binders sign (using an HMAC) the whole ClientHello message, including the handshake protocol header (the one byte handshake type and the 3-byte length), but excluding other binders.

tlsfuzzer.helpers.**psk_session_ext_gen**(*psk_settings=None*)
Generator that uses last New Session Ticket to create PSK extension.

Can optionally take a list of tuples that define static PSKs that will be added after the NST PSK. See *psk_ext_gen()* for description of their format.

Parameters **psk_settings** (*list*) – list of tuples

Returns extension generator

tlsfuzzer.helpers.**flexible_getattr**(*val*, *val_type*)
Convert a string of number, name, or None to object.

If the `val` is a number, return a number, when it's a string like `none` return `None` object. When it's a string representing one of the fields in provided type, return that value.

tlsfuzzer.helpers.**key_share_ext_gen**(*groups*)
Generator of key_share extension.

Generator that can be used to delay the generation of key shares for TLS 1.3 ClientHello.

Parameters **groups** (*list*) – TLS numerical IDs from GroupName identifying groups that should be present in the extension or ready to use KeyShareEntries.

Return type callable

tlsfuzzer.helpers.**uniqueness_check**(*values*, *count*)
Check if values in the lists in the dictionary are unique.

Also check if all the arrays have the length of `count`.

Parameters

- **values** – dictionary of lists to check
- **count** (*int*) – expected length of lists

Returns list of errors found

tlsfuzzer.helpers.**RSA_SIG_ALL = [(6, 1), (5, 1), (4, 1), (3, 1), (2, 1), (1, 1), (8, 4), (**
List of all RSA signature algorithms supported by tlsfuzzer, as used in `signature_algorithms` or `signature_algorithms_cert` extensions.

tlsfuzzer.helpers.**ECDSA_SIG_ALL = [(6, 3), (5, 3), (4, 3), (3, 3), (2, 3)]**
List of all ECDSA signature algorithms supported by tlsfuzzer, as used in `signature_algorithms` or `signature_algorithms_cert` extensions.

tlsfuzzer.helpers.**RSA_PKCS1_ALL = [(6, 1), (5, 1), (4, 1), (3, 1), (2, 1), (1, 1)]**
List of all signature algorithms that use PKCS#1 v1.5 padding.

tlsfuzzer.helpers.**RSA_PSS_PSS_ALL = [(8, 11), (8, 10), (8, 9)]**
List of all signature algorithms that use RSA-PSS padding and have been made with RSA-PSS key.

tlsfuzzer.helpers.**RSA_PSS_RSAE_ALL = [(8, 6), (8, 5), (8, 4)]**
List of all signature algorithms that use RSA-PSS padding and have been made with rsaEncryption (PKCS#1) key.

tlsfuzzer.helpers.**ECDSA_SIG_TLS1_3_ALL = [(6, 3), (5, 3), (4, 3)]**
List of all ECDSA signature algorithms that can be used in TLS 1.3.

Subset of *ECDSA_SIG_ALL*.

tlsfuzzer.helpers.**EDDSA_SIG_ALL = [(8, 8), (8, 7)]**
List of all EdDSA signature algorithms that can be used in TLS 1.2 and later.

tlsfuzzer.helpers.**SIG_ALL = [(8, 11), (8, 10), (8, 9), (8, 6), (8, 5), (8, 4), (6, 1), (5,**
List of all signature algorithms supported by tlsfuzzer, as used in `signature_algorithms` or `signature_algorithms_cert` extension.

For now includes only RSA, ECDSA and EdDSA algorithms, will include DSA algorithms later on.

Sorted in order of strongest to weakest hash.

class tlsfuzzer.helpers.**AutoEmptyExtension**
Bases: `object`

Identifier used to tell ClientHelloGenerator to create empty extension.

tlsfuzzer.helpers.**client_cert_types_to_ids**(*names*)
Convert a string with client certificate method names to list of IDs.

Parameters **names** (*str*) – whitespace separated list of names of client certificate types (used in CertificateRequest message in TLS 1.2 and earlier). Identifiers can be names (e.g. `rsa_sign`), or integers (e.g. `1` instead of `rsa_sign`).

Raises **AttributeError** – when the specified identifier is not defined in `ClientCertificateType`

Return type list of int

tlsfuzzer.messages module

Objects for generating TLS messages to send.

class tlsfuzzer.messages.**AlertGenerator**(*level=1*, *description=0*)
Bases: *tlsfuzzer.messages.MessageGenerator*

Generator for TLS Alert messages.

_repr(*attributes*)
Return a text representation of the object.

Parameters **attributes** (*list(str)*) – names of attributes of the object that will be included in the text representation

add_child(*child*)
Sets the parameter as the child of the node

Returns the child node

generate(*status*)
Send the Alert to server.

get_all_siblings()
Return iterator with all siblings of node

Return type iterator

is_command()
Define object as a generator node.

is_expect()
Define object as a generator node.

is_generator()
Define object as a generator node.

post_send(*state*)
Modify the state after sending the message.

class tlsfuzzer.messages.**ApplicationDataGenerator**(*payload*)
Bases: *tlsfuzzer.messages.MessageGenerator*

Generator for TLS Application Data messages.

_repr(*attributes*)
Return a text representation of the object.

Parameters **attributes** (*list(str)*) – names of attributes of the object that will be included in the text representation

add_child(*child*)
Sets the parameter as the child of the node

Returns the child node

generate(*status*)
Send data to server in Application Data messages.

get_all_siblings()
Return iterator with all siblings of node

Return type iterator

is_command()
Define object as a generator node.

is_expect()
Define object as a generator node.

is_generator()
Define object as a generator node.

post_send(*state*)
Modify the state after sending the message.

class tlsfuzzer.messages.**CertificateGenerator**(*certs=None*, *cert_type=None*, *version=None*, *context=None*)
Bases: *tlsfuzzer.messages.HandshakeProtocolMessageGenerator*

Generator for TLS handshake protocol Certificate message.

_repr(*attributes*)
Return a text representation of the object.

Parameters **attributes** (*list(str)*) – names of attributes of the object that will be included in the text representation

add_child(*child*)
Sets the parameter as the child of the node

Returns the child node

generate(*status*)
Create a Certificate message.

get_all_siblings()
Return iterator with all siblings of node

Return type iterator

is_command()
Define object as a generator node.

is_expect()
Define object as a generator node.

is_generator()
Define object as a generator node.

post_send(*state*)
Update handshake hashes after sending.

class tlsfuzzer.messages.**CertificateVerifyGenerator**(*private_key=None*, *msg_version=None*, *msg_alg=None*, *sig_version=None*, *sig_alg=None*, *signature=None*, *rsa_pss_salt_len=None*, *padding_xors=None*, *padding_subs=None*, *mgf1_hash=None*, *context=None*)
Bases: *tlsfuzzer.messages.HandshakeProtocolMessageGenerator*

Generator for TLS handshake protocol Certificate Verify message.

Variables

- **msg_alg** (*tuple(int,int)*) – signature and hash algorithm to be set on in the digitally-signed structure of TLSv1.2 Certificate Verify message. By default the first matching algorithm from CertificateRequest that matches our key or sent certificate. If no CertificateRequest received it will send the first algorithm matching our key or certificate sent. If no Certificate nor private key is available, it will select first algorithm from CertificateRequest. If no Certificate, CertificateRequest nor private key is availbale then it will use SHA-1 + RSA The first value in the tuple specifies hash type (from HashAlgorithm) and the second value specifies the signature algorithm (from SignatureAlgorithm). Or the value from SignatureScheme.
- **msg_version** (*tuple(int,int)*) – protocol version that the message is to use, default is taken from current connection state
- **sig_version** (*tuple(int,int)*) – protocol version to use for calculating the verify bytes for the signature (overrides msg_version, but just for the signature). Equal to msg_version by default.
- **sig_alg** (*tuple(int,int)*) – hash and signature algorithm to be used for creating the signature in the message. Equal to msg_alg by default. Requires the `sig_version` to be set to at least TLSv1.2 to be effective.
- **signature** (*bytearray*) – bytes to sent as the signature of the message
- **padding_xors** (*dict(int,int)*) – which bytes of the pre-encryption RSA padding or post-signature ECDSA signature should be xored and with what values
- **padding_subs** (*dict(int,int)*) – same as padding_xors but substitues specified bytes instead
- **mgf1_hash** (*str*) – name of the hash to be used for calculating MGF1, effective only if sig_alg is set to a RSA_PSS algorithm and sig_version is TLS 1.2 or greater. By default the hash taken from sig_alg.
- **rsa_pss_salt_len** (*int*) – length of the salt (in bytes) used in signature. Effective only if sig_alg is set to a RSA_PSS algorithm and sig_version is TLS 1.2 or greater. By default it's equal to the length of the hash taken from sig_alg.

- **private_key** (RSAKey or ECDSAKey) – key that will be used for signing the message

_get_ecdsa_sig_parameters()
Set up parameters for sign() operation with ecdsa keys.

_get_key_and_key_type(*status*)
Get a key, or if not possible, certificate for selecting the signature algorithm.

_get_rsa_sig_parameters()
Return parameters for sign() operation with rsa keys.

_make_signature(*status*)
Create signature for CertificateVerify message.

static _normalise_dict(*dictionary*, *max_byte*)

_normalise_subs_and_xors(*max_byte*)
Make sure that the substitutions and xors don't go over the size of buffer, this is fine as ECDSA signatures are ASN.1 objects so have variable size

_repr(*attributes*)
Return a text representation of the object.

Parameters **attributes** (*list(str)*) – names of attributes of the object that will be included in the text representation

_select_msg_alg(*status*)
Select the signature algorithm based on CertificateRequest from server, either our sent Certificate or our private key and the protocol version.

static _sig_alg_for_certificate(*key_alg*, *accept_sig_algs*, *version*, *key*)
Select an acceptable signature algorithm based on key algorithm, protocol version and curve name (in case of ECDSA).

static _sig_alg_for_ecdsa_key(*accept_sig_algs*, *version*, *key*)
Select an acceptable signature algorithm for a given ecdsa key.

static _sig_alg_for_eddsa_key(*key_alg*, *accept_sig_algs*)

static _sig_alg_for_rsa_key(*key_alg*, *accept_sig_algs*, *version*)
Select an acceptable signature algorithm for a given rsa key.

add_child(*child*)
Sets the parameter as the child of the node

Returns the child node

generate(*status*)
Create a CertificateVerify message.

get_all_siblings()
Return iterator with all siblings of node

Return type iterator

is_command()
Define object as a generator node.

is_expect()
Define object as a generator node.

is_generator()
Define object as a generator node.

post_send(*state*)
Update handshake hashes after sending.

class tlsfuzzer.messages.**ChangeCipherSpecGenerator**(*extended_master_secret=None*, *fake=False*)
Bases: *tlsfuzzer.messages.MessageGenerator*

Generator for TLS Change Cipher Spec messages.

Note: After sending the ChangeCipherSpec message, in TLS 1.2 and earlier, the record layer will switch to encrypted communication (or newly negotiated keys). In TLS 1.3 the message has no effect on encryption or record layer state.

_repr(*attributes*)
Return a text representation of the object.

Parameters **attributes** (*list(str)*) – names of attributes of the object that will be included in the text representation

add_child(*child*)
Sets the parameter as the child of the node

Returns the child node

generate(*status*)
Create a message for sending to server.

get_all_siblings()
Return iterator with all siblings of node

Return type iterator

is_command()
Define object as a generator node.

is_expect()
Define object as a generator node.

is_generator()
Define object as a generator node.

post_send(*status*)
Generate new encryption keys for connection.

class tlsfuzzer.messages.**ClearContext**(*context*)
Bases: *tlsfuzzer.messages.Command*

Object used to zero-out the context used in PHA.

This is necessary if the conversation is executed more than once.

_repr(*attributes*)
Return a text representation of the object.

Parameters **attributes** (*list(str)*) – names of attributes of the object that will be included in the text representation

add_child(*child*)
Sets the parameter as the child of the node

Returns the child node

get_all_siblings()
Return iterator with all siblings of node

Return type iterator

is_command()
Define object as a command node.

is_expect()
Define object as a command node.

is_generator()
Define object as a command node.

process(*state*)
Zero out the associated context

class tlsfuzzer.messages.**ClientHelloGenerator**(*ciphers=None*, *extensions=None*, *version=None*, *session_id=None*, *random=None*, *compression=None*, *ssl2=False*, *modifiers=None*)
Bases: *tlsfuzzer.messages.HandshakeProtocolMessageGenerator*

Generator for TLS handshake protocol Client Hello messages.

_generate_extensions(*state*)
Convert extension generators to extension objects.

_handle_modifiers(*state*, *clnt_hello*)
Handle processing of the modifiers of the message.

_repr(*attributes*)
Return a text representation of the object.

Parameters **attributes** (*list(str)*) – names of attributes of the object that will be included in the text representation

add_child(*child*)
Sets the parameter as the child of the node

Returns the child node

generate(*state*)
Create a Client Hello message.

get_all_siblings()
Return iterator with all siblings of node

Return type iterator

is_command()
Define object as a generator node.

is_expect()
Define object as a generator node.

is_generator()
Define object as a generator node.

post_send(*state*)
Update handshake hashes after sending.

class tlsfuzzer.messages.**ClientKeyExchangeGenerator**(*cipher=None*, *version=None*, *client_version=None*, *dh_Yc=None*, *padding_subs=None*, *padding_xors=None*, *ecdh_Yc=None*, *encrypted_premaster=None*, *modulus_as_encrypted_premaster=False*, *p_as_share=False*, *p_1_as_share=False*, *premaster_secret=None*, *padding_byte=None*, *reuse_encrypted_premaster=False*)

Bases: *tlsfuzzer.messages.HandshakeProtocolMessageGenerator*

Generator for TLS handshake protocol Client Key Exchange messages.

Variables

- **dh_Yc** (*int*) – Override the sent dh_Yc value to the specified one
- **padding_subs** (*dict(int,int)*) – Substitutions for the encrypted premaster secret padding bytes (applicable only for the RSA key exchange)
- **padding_xors** (*dict(int,int)*) – XORs for the encrypted premaster secret padding bytes (applicable only for the RSA key exchange)
- **ecdh_Yc** (*bytearray*) – encoded ECC point being the client key share for the key exchange
- **encrypted_premaster** (*bytearray*) – the premaster secret after it was encrypted, as it will be sent on the wire
- **modulus_as_encrypted_premaster** (*bool*) – if True, set the encrypted premaster (the value seen on the wire) to the server's certificate modulus (the server's public key)
- **p_as_share** (*bool*) – set the key share to the value p provided by server in Server Key Exchange (applicable only to FFDHE key exchange)
- **p_1_as_share** (*bool*) – set the key share to the value $p - 1$, as provided by server in Server Key Exchange (applicable only to FFDHE key exchange with safe primes)
- **padding_byte** (*int*) – byte to use as padding instead of randomly generated bytes (applicable only for RSA key exchange)
- **client_version** (*tuple(int,int)*) – the version to set in the RSA pre-master secret
- **reuse_encrypted_premaster** (*bool*) – if set to true, the message generator will create the RSA ciphertext once and reuse it for subsequent connections. Applicable only to RSA key exchange, useful only for tests that run the same conversation over and over (like timing tests).

_encrypt_with_fuzzing(*public_key*)

Use public_key to encrypt premaster secret with fuzzed padding.

_repr(*attributes*)

Return a text representation of the object.

Parameters **attributes** (*list(str)*) – names of attributes of the object that will be included in the text representation

add_child(*child*)
Sets the parameter as the child of the node

Returns the child node

generate(*status*)
Create a Client Key Exchange message.

get_all_siblings()
Return iterator with all siblings of node

Return type iterator

is_command()
Define object as a generator node.

is_expect()
Define object as a generator node.

is_generator()
Define object as a generator node.

post_send(*state*)
Save intermediate handshake hash value.

class tlsfuzzer.messages.**ClientMasterKeyGenerator**(*cipher=None*, *master_key=None*)
Bases: *tlsfuzzer.messages.HandshakeProtocolMessageGenerator*

Generator for SSLv2 Handshake Protocol CLIENT-MASTER-KEY message.

_repr(*attributes*)
Return a text representation of the object.

Parameters **attributes** (*list(str)*) – names of attributes of the object that will be included in the text representation

add_child(*child*)
Sets the parameter as the child of the node

Returns the child node

generate(*state*)
Generate a new CLIENT-MASTER-KEY message.

get_all_siblings()
Return iterator with all siblings of node

Return type iterator

is_command()
Define object as a generator node.

is_expect()
Define object as a generator node.

is_generator()
Define object as a generator node.

post_send(*state*)
Update handshake hashes after sending.

class tlsfuzzer.messages.**Close**
Bases: *tlsfuzzer.messages.Command*

Object used to close a TCP connection.

_repr(*attributes*)
Return a text representation of the object.

Parameters **attributes** (*list(str)*) – names of attributes of the object that will be included in the text representation

add_child(*child*)
Sets the parameter as the child of the node

Returns the child node

get_all_siblings()
Return iterator with all siblings of node

Return type iterator

is_command()
Define object as a command node.

is_expect()
Define object as a command node.

is_generator()
Define object as a command node.

process(*state*)
Close currently open connection.

class tlsfuzzer.messages.**CollectNonces**(*nonces*)
Bases: *tlsfuzzer.messages.Command*

Start collecting nonces being sent by the server in the provided array.

Works only for ciphers like AES-GCM which use explicit nonces. Ciphers like Chacha20 use implicit nonce constructed from PRF output and sequence number.

Needs to be run after the cipher was negotiated and switched to (after CCS), will collect nonces only till next renegotiation.

_repr(*attributes*)
Return a text representation of the object.

Parameters **attributes** (*list(str)*) – names of attributes of the object that will be included in the text representation

add_child(*child*)
Sets the parameter as the child of the node

Returns the child node

get_all_siblings()
Return iterator with all siblings of node

Return type iterator

is_command()
Define object as a command node.

is_expect()
Define object as a command node.

is_generator()
Define object as a command node.

process(*state*)
Monkey patch the seal() method.

class tlsfuzzer.messages.**Command**
Bases: *tlsfuzzer.tree.TreeNode*

Command objects.

_repr(*attributes*)
Return a text representation of the object.

Parameters **attributes** (*list(str)*) – names of attributes of the object that will be included in the text representation

add_child(*child*)
Sets the parameter as the child of the node

Returns the child node

get_all_siblings()
Return iterator with all siblings of node

Return type iterator

is_command()
Define object as a command node.

is_expect()
Define object as a command node.

is_generator()
Define object as a command node.

process(*state*)
Change the state of the connection.

class tlsfuzzer.messages.**Connect**(*hostname*, *port*, *version=(3, 0)*, *timeout=5*)
Bases: *tlsfuzzer.messages.Command*

Object used to connect to a TCP server.

_repr(*attributes*)
Return a text representation of the object.

Parameters **attributes** (*list(str)*) – names of attributes of the object that will be included in the text representation

add_child(*child*)
Sets the parameter as the child of the node

Returns the child node

get_all_siblings()
Return iterator with all siblings of node

Return type iterator

is_command()
Define object as a command node.

is_expect()
Define object as a command node.

is_generator()
Define object as a command node.

process(*state*)
Connect to a server.

class tlsfuzzer.messages.**CopyVariables**(*log*)
Bases: *tlsfuzzer.messages.Command*

Copy current random values of connection to provided arrays.

Available keys are either `ClientHello.random`, `ServerHello.random`, `ServerHello.session_id` or one of the values in `key` in *ConnectionState* (`premaster_secret`, `master_secret`, `ServerHello.extensions.key_share.key_exchange`, `server handshake traffic secret`, `exporter master secret`, `ServerKeyExchange.key_share`, `ServerKeyExchange.dh_p`, `DH shared secret`, `PSK secret`, `client_verify_data`, `server_verify_data`, `client application traffic secret`, `server application traffic secret`, `resumption master secret`, `early secret`, or `handshake secret`)

The log should be a dict (where keys have the above specified names) and values should be arrays (the values will be appended there).

This node needs to be put right after a node that calculate or use the specific values to guarantee correct collection (i.e. if the conversation performs a renegotiation, it needs to be placed after both *ExpectServerHello* nodes to collect both `ServerHello.random` values).

Parameters **log** (*dict(str,list)*) – dictionary with names of values to collect

_repr(*attributes*)
Return a text representation of the object.

Parameters **attributes** (*list(str)*) – names of attributes of the object that will be included in the text representation

add_child(*child*)
Sets the parameter as the child of the node

Returns the child node

get_all_siblings()
Return iterator with all siblings of node

Return type iterator

is_command()
Define object as a command node.

is_expect()
Define object as a command node.

is_generator()
Define object as a command node.

process(*state*)
Copy current variables to log arrays.

class tlsfuzzer.messages.**FinishedGenerator**(*protocol=None*, *trunc_start=0*, *trunc_end=None*, *pad_byte=0*, *pad_left=0*, *pad_right=0*, *context=None*)
Bases: *tlsfuzzer.messages.HandshakeProtocolMessageGenerator*

Generator for TLS handshake protocol Finished messages.

Note: The FinishedGenerator may influence the record layer encryption. In SSLv2, the record layer will be configured to expect encrypted records and send encrypted records *before* the message is sent. In SSLv3 up to

TLS 1.2 the message has no impact on state of encryption. In TLS 1.3, *after* the message is sent, the record layer will be switched to use `client_application_traffic_secret` keys for *sending*.

_repr(*attributes*)
Return a text representation of the object.

Parameters **attributes** (*list(str)*) – names of attributes of the object that will be included in the text representation

add_child(*child*)
Sets the parameter as the child of the node

Returns the child node

generate(*status*)
Create a Finished message.

get_all_siblings()
Return iterator with all siblings of node

Return type iterator

is_command()
Define object as a generator node.

is_expect()
Define object as a generator node.

is_generator()
Define object as a generator node.

post_send(*status*)
Perform post-transmit changes needed by generation of Finished.

class tlsfuzzer.messages.**FlushMessageList**(*fragment_list*)
Bases: *tlsfuzzer.messages.PopMessageFromList*

Takes a reference to list, empties it to generate a message.

_repr(*attributes*)
Return a text representation of the object.

Parameters **attributes** (*list(str)*) – names of attributes of the object that will be included in the text representation

add_child(*child*)
Sets the parameter as the child of the node

Returns the child node

generate(*state*)
Creata a single message to empty the list.

get_all_siblings()
Return iterator with all siblings of node

Return type iterator

is_command()
Define object as a generator node.

is_expect()
Define object as a generator node.

is_generator()
Define object as a generator node.

post_send(*state*)
Modify the state after sending the message.

class tlsfuzzer.messages.**HandshakeProtocolMessageGenerator**
Bases: *tlsfuzzer.messages.MessageGenerator*

Message generator for TLS Handshake protocol messages.

_repr(*attributes*)
Return a text representation of the object.

Parameters **attributes** (*list(str)*) – names of attributes of the object that will be included in the text representation

add_child(*child*)
Sets the parameter as the child of the node

Returns the child node

generate(*state*)
Return a message ready to write to socket.

get_all_siblings()
Return iterator with all siblings of node

Return type iterator

is_command()
Define object as a generator node.

is_expect()
Define object as a generator node.

is_generator()
Define object as a generator node.

post_send(*state*)
Update handshake hashes after sending.

class tlsfuzzer.messages.**HeartbeatGenerator**(*payload*, *message_type=1*, *padding_length=None*)
Bases: *tlsfuzzer.messages.MessageGenerator*

Generator for heartbeat messages.

Variables

- **message_type** (*int*) – the type of the message to send, see `HeartbeatMessageType` enum for values
- **payload** (*bytearray*) – data to be sent to the other size for it to echo it back
- **padding** (*bytearray*) – payload to be sent to the other side, it should be at least 16 bytes long for the message to be valid

_repr(*attributes*)
Return a text representation of the object.

Parameters **attributes** (*list(str)*) – names of attributes of the object that will be included in the text representation

add_child(*child*)
Sets the parameter as the child of the node

Returns the child node

generate(*state*)
Create a Heartbeat message.

Return type *~tlslite.messages.Heartbeat*

Returns heartbeat message to be sent to the other side

get_all_siblings()
Return iterator with all siblings of node

Return type iterator

is_command()
Define object as a generator node.

is_expect()
Define object as a generator node.

is_generator()
Define object as a generator node.

post_send(*state*)
Modify the state after sending the message.

class tlsfuzzer.messages.**KeyUpdateGenerator**(*message_type=0*)
Bases: *tlsfuzzer.messages.MessageGenerator*

Generator for TLS 1.3 KeyUpdate message.

_repr(*attributes*)
Return a text representation of the object.

Parameters **attributes** (*list(str)*) – names of attributes of the object that will be included in the text representation

add_child(*child*)
Sets the parameter as the child of the node

Returns the child node

generate(*state*)
Generate a KeyUpdate message.

get_all_siblings()
Return iterator with all siblings of node

Return type iterator

is_command()
Define object as a generator node.

is_expect()
Define object as a generator node.

is_generator()
Define object as a generator node.

post_send(*state*)
Perform post-transmit changes needed by generation of KeyUpdate.

class tlsfuzzer.messages.**MessageGenerator**
Bases: *tlsfuzzer.tree.TreeNode*

Message generator objects.

_repr(*attributes*)
Return a text representation of the object.

Parameters **attributes** (*list(str)*) – names of attributes of the object that will be included in the text representation

add_child(*child*)
Sets the parameter as the child of the node

Returns the child node

generate(*state*)
Return a message ready to write to socket.

get_all_siblings()
Return iterator with all siblings of node

Return type iterator

is_command()
Define object as a generator node.

is_expect()
Define object as a generator node.

is_generator()
Define object as a generator node.

post_send(*state*)
Modify the state after sending the message.

class tlsfuzzer.messages.**PlaintextMessageGenerator**(*content_type*, *data*, *description=None*)
Bases: *tlsfuzzer.messages.Command*

Send a plaintext data record irrespective of encryption state.

Does not update handshake hashes, record layer state, does not fragment, etc.

Variables

- **content_type** (*int*) – content type of message, used in record layer header. See ContentType for well-known values
- **data** (*bytearray*) – payload for the record
- **description** (*str*) – identifier to print when processing of the node fails

_repr(*attributes*)
Return a text representation of the object.

Parameters **attributes** (*list(str)*) – names of attributes of the object that will be included in the text representation

add_child(*child*)
Sets the parameter as the child of the node

Returns the child node

get_all_siblings()
Return iterator with all siblings of node

Return type iterator

is_command()
Define object as a command node.

is_expect()
Define object as a command node.

is_generator()
Define object as a command node.

process(*state*)
Send the message over the socket.

class tlsfuzzer.messages.**PopMessageFromList**(*fragment_list*)
Bases: *tlsfuzzer.messages.MessageGenerator*

Takes a reference to list, pops a message from it to generate one.

_repr(*attributes*)
Return a text representation of the object.

Parameters **attributes** (*list(str)*) – names of attributes of the object that will be included in the text representation

add_child(*child*)
Sets the parameter as the child of the node

Returns the child node

generate(*state*)
Create a message using the reference to list from init.

get_all_siblings()
Return iterator with all siblings of node

Return type iterator

is_command()
Define object as a generator node.

is_expect()
Define object as a generator node.

is_generator()
Define object as a generator node.

post_send(*state*)
Modify the state after sending the message.

class tlsfuzzer.messages.**RawMessageGenerator**(*content_type*, *data*, *description=None*)
Bases: *tlsfuzzer.messages.MessageGenerator*

Generator for arbitrary record layer messages.

Can generate message with any content_type and any payload. Will be encrypted if encryption is negotiated in the connection.

Variables

- **content_type** (*int*) – content type of message, used in record layer header. See ContentType for well-known values
- **data** (*bytearray*) – payload for the record
- **description** (*str*) – identifier to print when processing of the node fails

_repr(*attributes*)
Return a text representation of the object.

Parameters **attributes** (*list(str)*) – names of attributes of the object that will be included in the text representation

add_child(*child*)
Sets the parameter as the child of the node

Returns the child node

generate(*state*)
Create a tlslite-ng message that can be send.

get_all_siblings()
Return iterator with all siblings of node

Return type iterator

is_command()
Define object as a generator node.

is_expect()
Define object as a generator node.

is_generator()
Define object as a generator node.

post_send(*state*)
Modify the state after sending the message.

class tlsfuzzer.messages.**RawSocketWriteGenerator**(*data*, *description=None*)
Bases: *tlsfuzzer.messages.Command*

Send a plaintext data irrespective of encryption state.

Does not update handshake hashes, record layer state, does not fragment, etc.

Variables

- **data** (*bytearray*) – data to send
- **description** (*str*) – identifier to print when processing of the node fails

_repr(*attributes*)
Return a text representation of the object.

Parameters **attributes** (*list(str)*) – names of attributes of the object that will be included in the text representation

add_child(*child*)
Sets the parameter as the child of the node

Returns the child node

get_all_siblings()
Return iterator with all siblings of node

Return type iterator

is_command()
Define object as a command node.

is_expect()
Define object as a command node.

is_generator()
Define object as a command node.

process(*state*)
Send the message over the socket.

class tlsfuzzer.messages.**ResetHandshakeHashes**
Bases: *tlsfuzzer.messages.Command*

Object used to reset current state of handshake hashes to zero.

Used for session renegotiation or resumption.

Also prepares for negotiation (or dropping) of record_size_limit extension.

_repr(*attributes*)
Return a text representation of the object.

Parameters **attributes** (*list(str)*) – names of attributes of the object that will be included in the text representation

add_child(*child*)
Sets the parameter as the child of the node

Returns the child node

get_all_siblings()
Return iterator with all siblings of node

Return type iterator

is_command()
Define object as a command node.

is_expect()
Define object as a command node.

is_generator()
Define object as a command node.

process(*state*)
Reset current running handshake protocol hashes.

class tlsfuzzer.messages.**ResetRenegotiationInfo**(*client=None*, *server=None*)
Bases: *tlsfuzzer.messages.Command*

Object used to reset state of data needed for secure renegotiation.

_repr(*attributes*)
Return a text representation of the object.

Parameters **attributes** (*list(str)*) – names of attributes of the object that will be included in the text representation

add_child(*child*)
Sets the parameter as the child of the node

Returns the child node

get_all_siblings()
Return iterator with all siblings of node

Return type iterator

is_command()
Define object as a command node.

is_expect()
Define object as a command node.

is_generator()
Define object as a command node.

process(*state*)
Reset current Finished message values.

class tlsfuzzer.messages.**ResetWriteConnectionState**
Bases: *tlsfuzzer.messages.Command*

Reset _writeState configuration to default values

All sent messages will be unencrypted now

_repr(*attributes*)
Return a text representation of the object.

Parameters **attributes** (*list(str)*) – names of attributes of the object that will be included in the text representation

add_child(*child*)
Sets the parameter as the child of the node

Returns the child node

get_all_siblings()
Return iterator with all siblings of node

Return type iterator

is_command()
Define object as a command node.

is_expect()
Define object as a command node.

is_generator()
Define object as a command node.

process(*state*)
Change the state of the connection.

class tlsfuzzer.messages.**SetMaxRecordSize**(*max_size=None*)
Bases: *tlsfuzzer.messages.Command*

Change the Record Layer to send records of non standard size.

_repr(*attributes*)
Return a text representation of the object.

Parameters **attributes** (*list(str)*) – names of attributes of the object that will be included in the text representation

add_child(*child*)
Sets the parameter as the child of the node

Returns the child node

get_all_siblings()
Return iterator with all siblings of node

Return type iterator

is_command()
Define object as a command node.

is_expect()
Define object as a command node.

is_generator()
Define object as a command node.

process(*state*)
Change the size of messages in record layer.

class tlsfuzzer.messages.**SetPaddingCallback**(*cb=None*)
Bases: *tlsfuzzer.messages.Command*

Set the padding callback which returns the length of the padding to be added to the message in the record layer.

_repr(*attributes*)
Return a text representation of the object.

Parameters **attributes** (*list(str)*) – names of attributes of the object that will be included in the text representation

add_child(*child*)
Sets the parameter as the child of the node

Returns the child node

static add_fixed_padding_cb(*size*)
Returns a callback function which returns a fixed number as the padding size

static fill_padding_cb(*length*, *contenttype*, *max_padding*)
Simple callback which returns the maximum padding size as the size of the padding to be added to the message

static fixed_length_cb(*size*)
Returns a callback function which returns a fixed number as the padding size

get_all_siblings()
Return iterator with all siblings of node

Return type iterator

is_command()
Define object as a command node.

is_expect()
Define object as a command node.

is_generator()
Define object as a command node.

process(*state*)
Set the callback which returns the length of the padding in record layer.

class tlsfuzzer.messages.**SetRecordVersion**(*version*)
Bases: *tlsfuzzer.messages.Command*

Change the version used at record layer.

_repr(*attributes*)
Return a text representation of the object.

Parameters **attributes** (*list(str)*) – names of attributes of the object that will be included in the text representation

add_child(*child*)
Sets the parameter as the child of the node

Returns the child node

get_all_siblings()
Return iterator with all siblings of node

Return type iterator

is_command()
Define object as a command node.

is_expect()
Define object as a command node.

is_generator()
Define object as a command node.

process(*state*)
Change the state of the connection.

class tlsfuzzer.messages.**TCPBufferingDisable**
Bases: *tlsfuzzer.messages.Command*

Stop buffering all writes on the TCP level.

All messages will be now passed directly to the TCP socket

_repr(*attributes*)
Return a text representation of the object.

Parameters **attributes** (*list(str)*) – names of attributes of the object that will be included in the text representation

add_child(*child*)
Sets the parameter as the child of the node

Returns the child node

get_all_siblings()
Return iterator with all siblings of node

Return type iterator

is_command()
Define object as a command node.

is_expect()
Define object as a command node.

is_generator()
Define object as a command node.

process(*state*)
Disable TCP buffering.

class tlsfuzzer.messages.**TCPBufferingEnable**
Bases: *tlsfuzzer.messages.Command*

Start buffering all writes on the TCP level of connection.

You will need to call an explicit flush to send the messages.

_repr(*attributes*)
Return a text representation of the object.

Parameters **attributes** (*list(str)*) – names of attributes of the object that will be included in the text representation

add_child(*child*)
Sets the parameter as the child of the node

Returns the child node

get_all_siblings()
Return iterator with all siblings of node

Return type iterator

is_command()
Define object as a command node.

is_expect()
Define object as a command node.

is_generator()
Define object as a command node.

process(*state*)
Enable TCP buffering.

class tlsfuzzer.messages.**TCPBufferingFlush**
Bases: *tlsfuzzer.messages.Command*

Send all messages in the buffer.

Does not change the state of buffering

_repr(*attributes*)
Return a text representation of the object.

Parameters **attributes** (*list(str)*) – names of attributes of the object that will be included in the text representation

add_child(*child*)
Sets the parameter as the child of the node

Returns the child node

get_all_siblings()
Return iterator with all siblings of node

Return type iterator

is_command()
Define object as a command node.

is_expect()
Define object as a command node.

is_generator()
Define object as a command node.

process(*state*)
Flush all messages to TCP socket.

tlsfuzzer.messages.**ch_cookie_handler**(*state*)
Client Hello cookie extension handler.

Copies the cookie extension from last HRR message.

tlsfuzzer.messages.**ch_key_share_handler**(*state*)
Client Hello key_share extension handler.

Generates the key share for the group selected by server in the last HRR message.

tlsfuzzer.messages.**div_ceil**(*divident*, *divisor*)
Perform integer division of divident by divisor, round up.

tlsfuzzer.messages.**fuzz_encrypted_message**(*generator*, *substitutions=None*, *xors=None*)
Change arbitrary bytes of the authenticated ciphertext block.

Can modify authentication tag of AEAD ciphers and CBC ciphers working in encrypt then MAC mode.

Parameters

- **generator** (MessageGenerator) – modified message
- **substitutions** (*dict(int,int)*) – modify specified bytes of the message, the keys indicate the positions in the message (negative numbers count from the end of messages), the values of the dictionary specify the values to change the bytes to
- **xors** (*dict(int,int)*) – modify specified bytes of the message, the keys indicate the positions in the message (negative numbers count from the end of messages), the values of the dictionary specify the values to xor with

tlsfuzzer.messages.**fuzz_mac**(*generator*, *substitutions=None*, *xors=None*)
Change arbitrary bytes of the MAC value.

Works with stream and CBC cipher suites in SSL 3 up to TLS 1.2. Works with both encrypt then MAC and MAC then encrypt connections.

Parameters

- **generator** (MessageGenerator) – modified message
- **substitutions** (*dict(int,int)*) – modify specified bytes of the message, the keys indicate the positions in the message (negative numbers count from the end of messages), the values of the dictionary specify the values to change the bytes to
- **xors** (*dict(int,int)*) – modify specified bytes of the message, the keys indicate the positions in the message (negative numbers count from the end of messages), the values of the dictionary specify the values to xor with

tlsfuzzer.messages.**fuzz_message**(*generator*, *substitutions=None*, *xors=None*)
Change arbitrary bytes of the message after write.

Modified data includes handshake protocol header but doesn't include record header, content type or record-level padding.

Parameters

- **generator** (MessageGenerator) – modified message
- **substitutions** (*dict(int,int)*) – modify specified bytes of the message, the keys indicate the positions in the message (negative numbers count from the end of messages), the values of the dictionary specify the values to change the bytes to
- **xors** (*dict(int,int)*) – modify specified bytes of the message, the keys indicate the positions in the message (negative numbers count from the end of messages), the values of the dictionary specify the values to xor with

tlsfuzzer.messages.**fuzz_padding**(*generator*, *min_length=None*, *substitutions=None*, *xors=None*)
Change the padding of the message.

Works with CBC ciphers only.

Note: the "-1" position is the byte with the length of padding while "-2" is the last byte of padding (if padding has non-zero length)

Parameters

- **generator** (MessageGenerator) – modified message
- **min_length** (*int*) – the minimum length of padding created, including the byte specifying length of padding, must be smaller than 257
- **substitutions** (*dict(int,int)*) – modify specified bytes of the message, the keys indicate the positions in the message (negative numbers count from the end of messages), the values of the dictionary specify the values to change the bytes to
- **xors** (*dict(int,int)*) – modify specified bytes of the message, the keys indicate the positions in the message (negative numbers count from the end of messages), the values of the dictionary specify the values to xor with

tlsfuzzer.messages.**fuzz_pkcs1_padding**(*key*, *substitutions=None*, *xors=None*, *padding_byte=None*)

Fuzz the PKCS#1 padding used in signatures or encryption.

Use to modify Client Key Exchange padding of encrypted value.

tlsfuzzer.messages.**fuzz_plaintext**(*generator*, *substitutions=None*, *xors=None*)

Change arbitrary bytes of the plaintext right before encryption.

Get access to all data before encryption, including the IV, MAC and padding.

Works only with CBC ciphers. in EtM mode will not include MAC.

Note: the "-1" position is the byte with length of padding while "-2" is the last byte of padding (if padding has non-zero length)

Parameters

- **substitutions** (*dict(int,int)*) – modify specified bytes of the message, the keys indicate the positions in the message (negative numbers count from the end of messages), the values of the dictionary specify the values to change the bytes to
- **xors** (*dict(int,int)*) – modify specified bytes of the message, the keys indicate the positions in the message (negative numbers count from the end of messages), the values of the dictionary specify the values to xor with

tlsfuzzer.messages.**pad_handshake**(*generator*, *size=0*, *pad_byte=0*, *pad=None*)

Pad or truncate handshake messages.

Pad or truncate a handshake message by given amount of bytes, use negative size to truncate. Update handshake protocol header to compensate.

Parameters

- **generator** (MessageGenerator) – modified message
- **size** (*int*) – number of bytes to add at the end (if positive) or number of bytes to remove at the end of payload (if negative)
- **pad_byte** (*int*) – numerical value of added bytes, must be between 0 and 255 inclusive
- **pad** (*bytearray*) – bytes to add at the end of payload

tlsfuzzer.messages.**post_send_msg_sock_restore**(*obj*, *method_name*, *old_method_name*)

Un-Monkey patch a method of msg_sock.

(Method used internally by tlsfuzzer.)

tlsfuzzer.messages.**replace_plaintext**(*generator*, *new_plaintext*)

Change the plaintext of the message right before encryption.

Will replace all data before encryption, including the IV, MAC and padding.

Note: works only with CBC ciphers. in EtM mode will NOT modify MAC.

Length of new_plaintext must be multiple of negotiated cipher block size (8 bytes for 3DES, 16 bytes for AES)

tlsfuzzer.messages.**split_message**(*generator*, *fragment_list*, *size*)
Split a given message type to multiple messages.

Allows for splicing message into the middle of a different message type

tlsfuzzer.messages.**substitute_and_xor**(*data*, *substitutions*, *xors*)
Apply changes from substitutions and xors to data for fuzzing.

(Method used internally by tlsfuzzer.)

tlsfuzzer.messages.**truncate_handshake**(*generator*, *size=0*, *pad_byte=0*)
Truncate a handshake message.

See *pad_handshake()* for inverse of this function

tlsfuzzer.runner module

Main event loop for running test cases

class tlsfuzzer.runner.**ConnectionState**
Bases: object

Keeps the TLS connection state for sending of messages

Variables

- **msg_sock** (*MessageSocket*) – message level abstraction for TLS Record Socket
- **handshake_hashes** – all handshake messages hashed
- **handshake_messages** – all hadshake messages exchanged between peers
- **key** – various computed cryptographic keys, hashes and secrets related to handshake and record layer

  premaster_secret - premaster secret from TLS 1.2 and earlier

  client finished handshake hashes - HandshakeHashes object that has the handshake hashes of last handshake (the only Handshake in TLS 1.3) up to and including the client Finished; used for post-handshake authentication

get_last_message_of_type(*msg_type*)
Returns last handshake message of provided type

get_server_public_key()
Extract server public key from server Certificate message

prf_name
Return the name of the PRF used for session.

TLS 1.3 specific function

prf_size
Return the size of the PRF output used for session.

TLS 1.3 specific function

class tlsfuzzer.runner.**Runner**(*conversation*)
Bases: object

Test if sending a set of commands returns expected values

run()
Execute conversation

tlsfuzzer.runner.**guess_response**(*content_type*, *data*, *ssl2=False*)
Guess which kind of message is in the record layer payload

tlsfuzzer.scanner module

class tlsfuzzer.scanner.**Fingerprint**(*ip*, *port*)
Bases: object

class tlsfuzzer.scanner.**Scanner**
Bases: object

scan(*ip=None*, *port=443*)

tlsfuzzer.timing_runner module

Tooling for running tests repeatedly

class tlsfuzzer.timing_runner.**TimingRunner**(*name*, *tests*, *out_dir*, *ip_address*, *port*, *interface*, *affinity=None*)
Bases: object

Repeatedly runs tests and captures timing information.

static _format_seconds(*sec*)
Format number of seconds into a more readable string.

static _report_progress(*status*)
Periodically report progress of task in status, thread runner.

status must be an array with three elements, first two specify a fraction of completed work (i.e. 0 <= status[0]/status[1] <= 1), third specifies if the reporting process should continue running, a False value there will cause the process to finish

analyse()
Starts analysis if available

Returns int 0 for no difference, 1 for difference, 2 unavailable

static check_analysis_availability()
Checks if additional packages are installed so analysis can run.

Returns bool Indicating if it is okay to run

static check_extraction_availability()
Checks if additional packages are installed so extraction can run.

Returns bool Indicating if it is okay to run

static check_tcpdump()
Checks if tcpdump is installed.

Returns boolean value indicating if tcpdump is present

create_output_directory(*name*)
Creates a new directory in the specified path to store results in.

Parameters **name** (*str*) – Name of the test being run

Returns str Path to newly created directory

extract()
Starts the extraction if available.

generate_log(*run_only*, *run_exclude*, *repetitions*)
Creates log with number of requested shuffled runs. :param set run_only: List of tests to be run exclusively :param set run_exclude: List of tests to exclude :param int repetitions: How many times to repeat each test

run()
Run test the specified number of times and start analysis

Returns int 0 for no difference, 1 for difference, 2 if unavailable

sniff()
Start tcpdump with filter on communication to/from server

tcpdump_status(*process*)
Checks if tcpdump is running. Intended to be run as a separate thread.

Parameters **process** (*Popen*) – A process with running tcpdump attached

tlsfuzzer.tree module

Handling of event tree nodes

class tlsfuzzer.tree.**TreeNode**
Bases: object

Base class for decision tree objects.

_repr(*attributes*)
Return a text representation of the object.

Parameters **attributes** (*list(str)*) – names of attributes of the object that will be included in the text representation

add_child(*child*)
Sets the parameter as the child of the node

Returns the child node

get_all_siblings()
Return iterator with all siblings of node

Return type iterator

is_command()
Checks if the object is a standalone state modifier

Return type bool

is_expect()
Checks if the object is a node which processes messages

Return type bool

is_generator()
Checks if the object is a generator for messages to send

Return type bool

CHAPTER 15

Indices and tables

- genindex
- modindex
- *Glossary*
- search

Python Module Index

t

Index

Symbols

A

C

D

E

F

G

K

M

N

P

R

S

T

U