DTLS Multicast Test Plan

# Testing Overview

The wolfSSL project uses several types of testing.

## Build and Unit Testing

The build process provides a suite of tests that are run when making the target “check”. They provide an initial and quick correctness test.

These tests are:

* unit – performs a unit test on most APIs, runs the known answer test on all enabled cryptography algorithms, and performs connection tests using every enabled cipher suite with blocking and non-blocking sockets, SSL version, and DTLS
* testsuite – performs a known answer test on all enabled cryptography algorithms and then makes a thread-to-thread TLS connection
* resume – test script that starts up the example server and uses the client to perform a connection and then resume the session
* external – uses the example client to connect to the wolfSSL website via HTTPS
* Google – uses the example client to connect to www.google.com via HTTPS
* OpenSSL – interoperation test using the OpenSSL s\_server and s\_client and the wolfSSL example client and server

The autoconf build we use turns on as many of the compiler options for additional warnings and checks we can. We make sure the code also compiles cleanly without warnings using Visual Studio on Windows, clang on macOS, gcc on Ubuntu, and several embedded IDEs.

## Static Analysis

We perform static analysis primarily using clang’s scan-build tool. We also use Facebook’s static analyzer, Infer. Every couple of months we run our code base through Coverity’s static analysis tool; customers will share their reports from Coverity when they use it as well.

## Memory Testing

On Ubuntu, we run memory testing using Valgrind. We will test the primary configuration for a project to check for leaks or bad reads and writes. All reports are investigated.

## Fuzz Testing

wolfSSL is testing using two different fuzz testing tools. One links into the library and pseudorandomly modifies parts of messages to attempt to trigger errors. The other sits in between a client and server and overwrites packets sequentially, one byte at a time, using several values. The goal for the testing is to attempt to induce a crash based on the tainted data.

## Peer Review

All code changes are peer reviewed before being accepted into the main repository.

# Automated Testing

Every pull request made to our GitHub repository is automatically tested by our Jenkins based test server. Many of the above tests are performed as is a test consisting of combinations of enabled options and disabled options. Additionally, when the test server is idle from pull request tests the fuzz test runs continuously. In addition to the little-endian Intel based test machine, a PowerPC based big-endian machine is also used in the testing.

# DTLS Multicast Project Specific Testing

The DTLS Multicast specific behavior has its own specialized testing needs.

## Missed and Delayed Packets

DTLS is tolerant of missed, delayed, and duplicated datagrams due to the nature of the UDP. Application protocols need to be tolerant of missed and delayed datagrams as well. DTLS handshakes are not tolerant of missed packets, but the handshake has its own retry mechanism. DTLS also filters out duplicate packets. wolfSSL (the company not the library) has a test tool for tampering with the datagrams. It sits as a man in the middle between two nodes and can modify datagrams based on command line options. It will be used to drop particular handshake messages during a node contacting the floating master for the keys. Also, it can drop out application packets during run time.

## Load Testing

Full load testing will not be possible in the lab, there are at best 10 devices to load test with. While the Multicast DTLS layer must be able to support 100 nodes, it must also do its normal work as well. A test tool will attempt to flood a peer while pretending to be 99 other nodes. The proxy that drops datagrams will be added in to attempt to confuse things, as well.

The testing roll out plan is to test first on macOS (Darwin/BSD), followed by Ubuntu (GNU/Linux), and finally the target platform. Message loading will progress from 1 datagram per second to 50 datagrams per second. (1, 2, 5, 10, 20, 50)