

Testing TLS

Hubert Kario
Quality Engineer
24-10-2015



2014

“Few” things happened last year. In short: every big cryptographic library had some critical flaws.

Heartbleed

24-10-2015

3/56



we had a bugs in handling of obscure TLS feature that exposed private keys

OpenSSL CCS bug

24-10-2015

4/56



we had a bug in handling state machine transitions which caused the encryption to use predictable keys

gotofail

we had regular typos that caused lack of verification of signatures

Certificate handling

24-10-2015

6/56



we also had NSS accepting invalid signatures in certificates and GnuTLS accepting non CA certs as CA certs

CVE-2014-6321 in schannel a.k.a. Winshock

24-10-2015

7/56



we had a remote code execution in Windows caused by incorrect message parsing

POODLE

24-10-2015

8/56



And the year ended with the final nail to SSLv3 coffin.



2015

Next year wasn't much better, but we discovered problems with protocol and misconfigured servers rather than implementations.

FREAK

24-10-2015

10/56



Export grade keys got factored

LOGJAM

24-10-2015

11/56



And small Diffie Hellman parameters were broken.

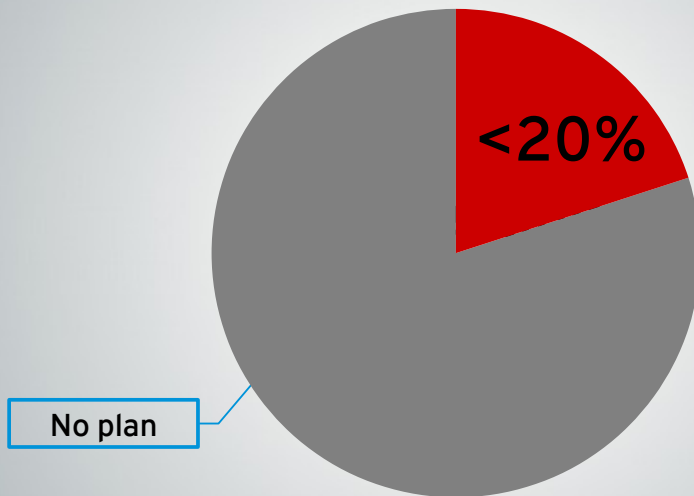


State of testing

So, how did this happen? After all, those are all security critical libraries, they are tested, aren't they?!

Well, “tested”, yes, tested well, not necessarily.

OSS projects w/test plans

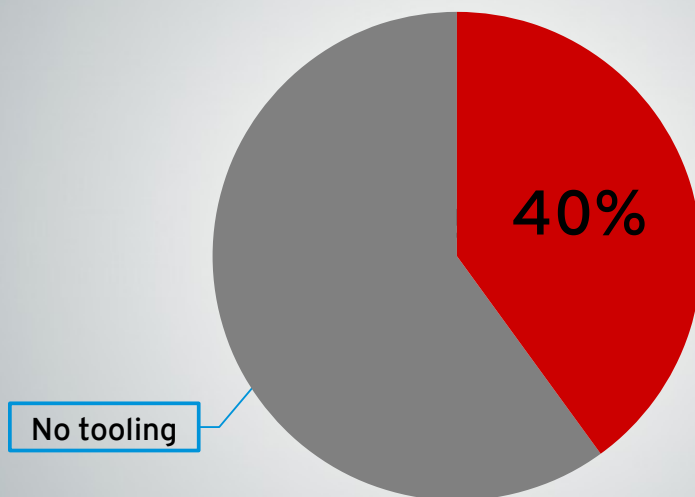


Source: Farooq & Quadri, 2011

Open Source projects in general don't use test plans.

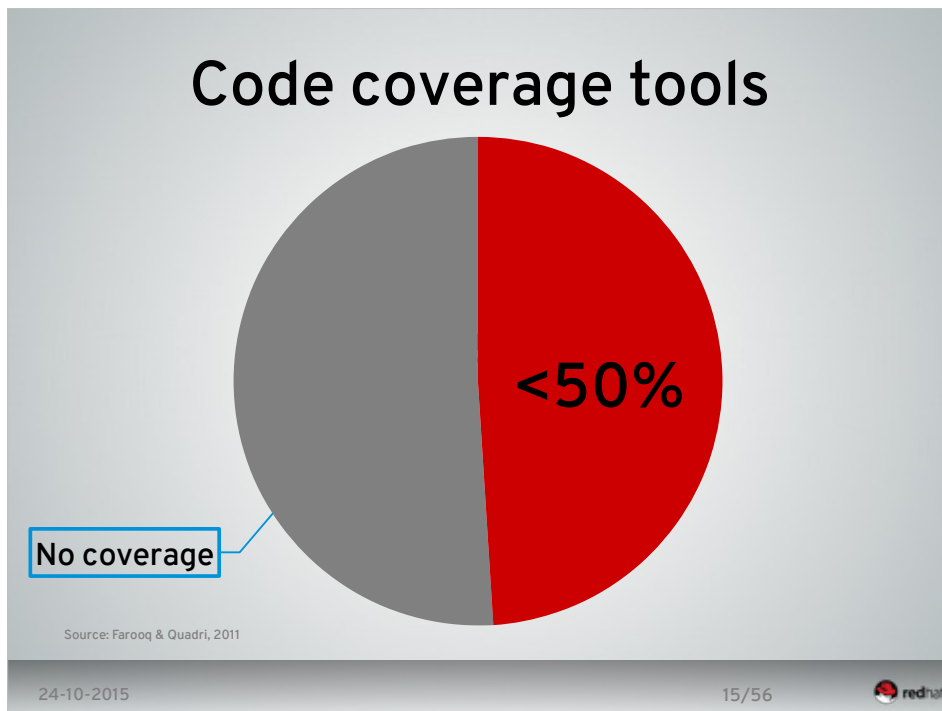
Farooq & Quadri, 2011. "Empirical evaluation of software testing techniques in an open source fashion"

OSS projects w/test tools

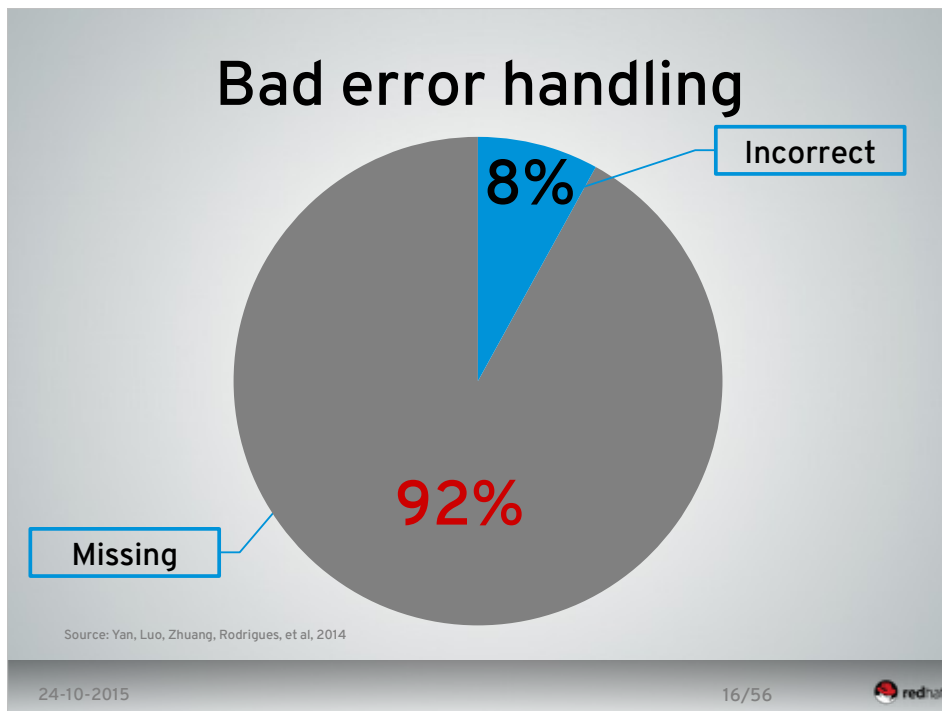


Source: Farooq & Quadri, 2011

About 40% use any kind of standard tooling for conducting tests.
Farooq & Quadri, 2011.



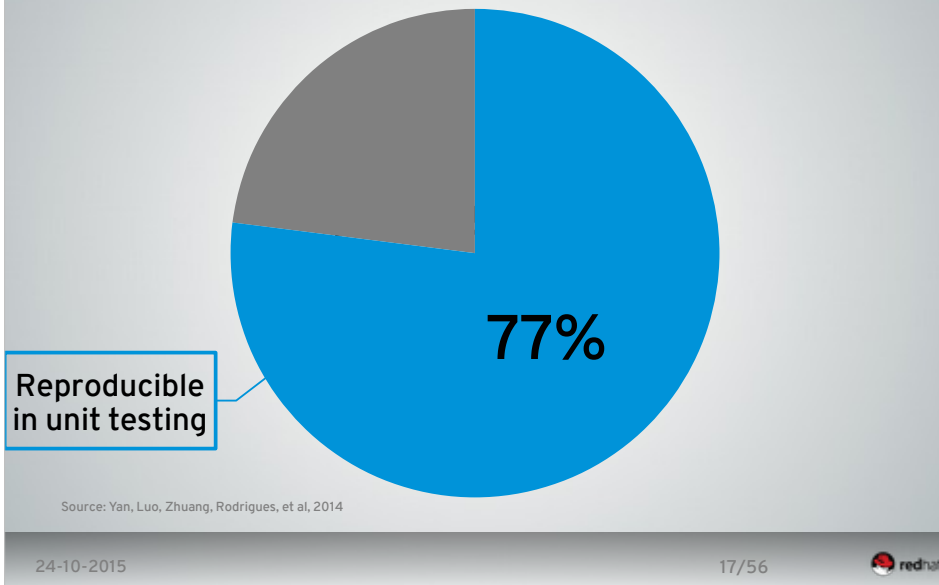
Less than 50% use code coverage measurement
Farooq & Quadri, 2011.



It's important because a lot of critical bugs are caused by bad error handling.

Yan, Luo, Zhuang, Rodrigues, et al, 2014. "Simple Testing Can Prevent Most Critical Failures: An Analysis of Production Failures in Distributed Data-Intensive Systems"

Unit tests vs bugs



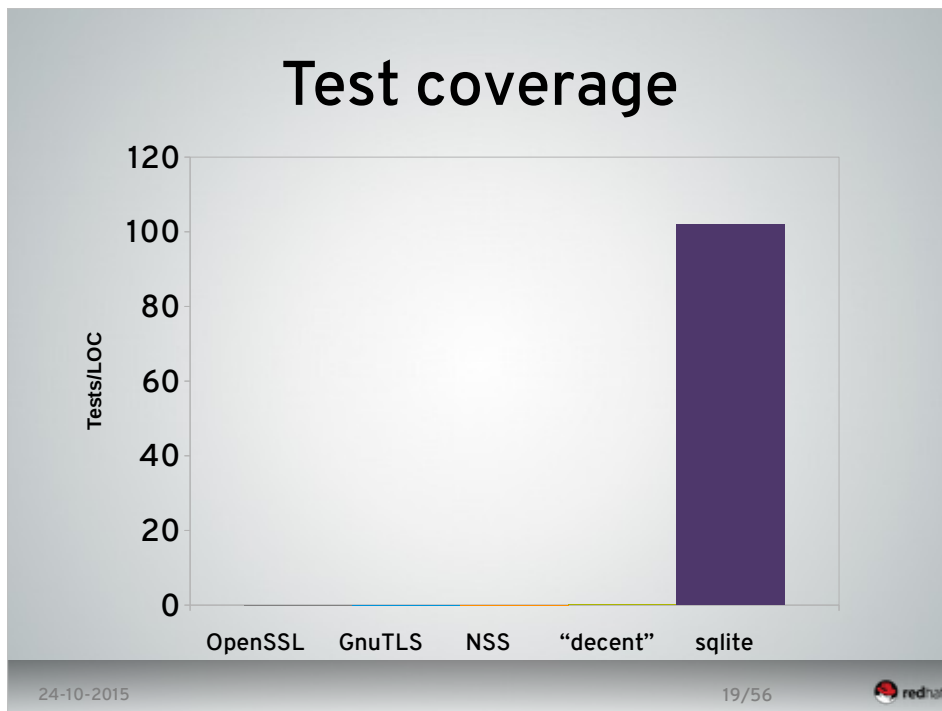
At the same time most of those are reproducible inside unit test cases, those that are not is usually because the project didn't have unit testing

Yan, Luo, Zhuang, Rodrigues, et al, 2014. .

OSS TLS libraries

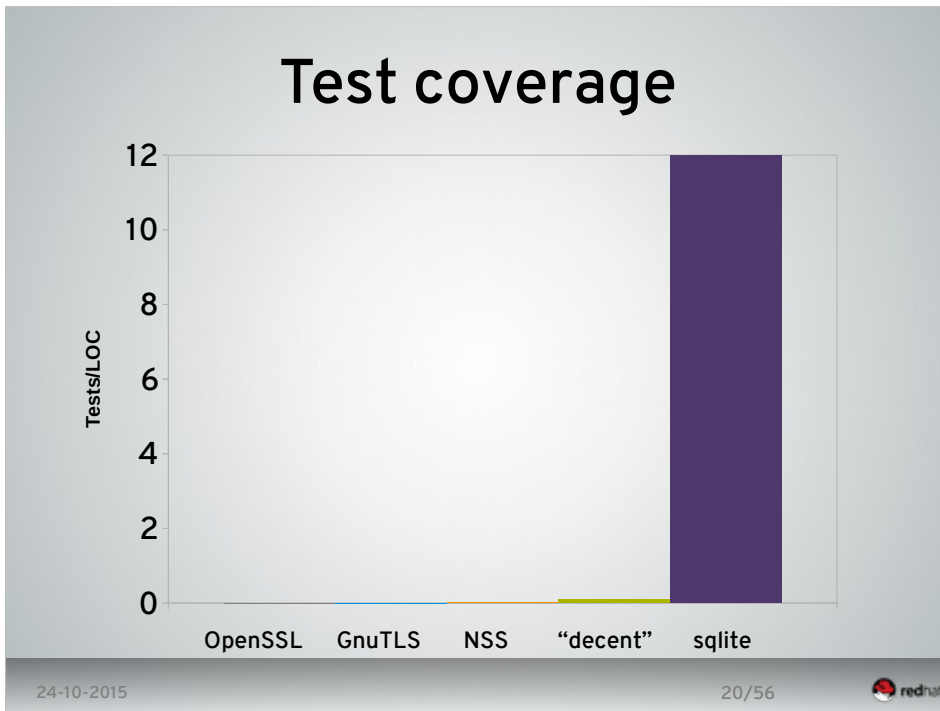
	OpenSSL	NSS	GnuTLS
Framework			
N° tests	100-200	>7000	100-200
Negative tests			

How do OSS TLS libraries compare? Not well. OpenSSL doesn't follow good testing practice and has minimal test coverage. GnuTLS only adds use of standard test framework to that. NSS may look good here, but not if we compare it to other pieces of code.

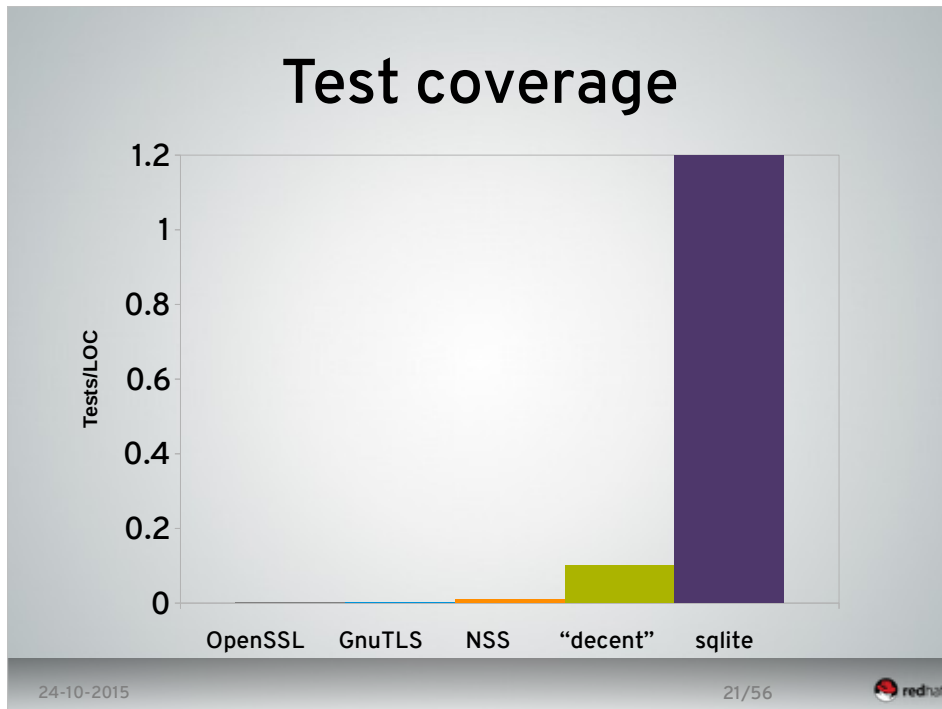


So let's compare the test coverage to other systems. The scale on the left is the ratio of number of test cases to lines of code. Sqlite has 97k LOC and 10M test cases so has ratio of 103.

Hmm, doesn't look like our libraries register on the scale... let's zoom a bit



Still hard to see... let's zoom by a factor of ten again.



Ahh, there we go!

What I'd call decent test coverage is one test case for every 3 to 8 lines of code, the minimum that is necessary to get to 100% branch coverage in testing.

(NSS has about 670kLOC and 7k test cases so about 0.01 test cases per line of code.

OpenSSL and GnuTLS both have about 500kLOC so they don't register on the scale still, we would have to zoom 3 times more for them to be visible.)



So why is that security libraries don't perform more testing?

Why the testing they undergone till now (including by other researchers) have had limited effect?

Libraries and bad data

24-10-2015

23/56



The libraries we have are primarily written to produce correct output. That means it is hard to make them send invalid data. That makes it in turn hard to create negative tests. So you either need to have a parallel implementation of TLS just for testing or to hack the main library and add ways that make it misbehave. Both rather unattractive.

Invisible bugs

Implementation can leak good deal of information through timing or how exactly it handles given failures. Not checking if a value sent by the other side is correct if all the other implementations send only good data can remain undetected for a long time. Which leads us to...

Fuzzy testing

24-10-2015

25/56



...fuzzy testing. Problem with TLS is that it has much more complex grammar compared to typical ASCII based protocols like SMTP or HTTP. Additionally, typical network fuzzers focus on creating inputs that crash the server, not ones that will allow you to extract private keys from the server.

Finally, to test TLS you need to implement full TLS and its cryptography – in some situations you could say that TLS is used as a transport layer for TLS.

Compatibility fears

24-10-2015

26/56



From the other side, we have administrators which are afraid of changes as they may break older clients.

Fears of untested code

24-10-2015

27/56



And in general, there's fear of deploying code with limited tests because of possible bugs lurking in it.

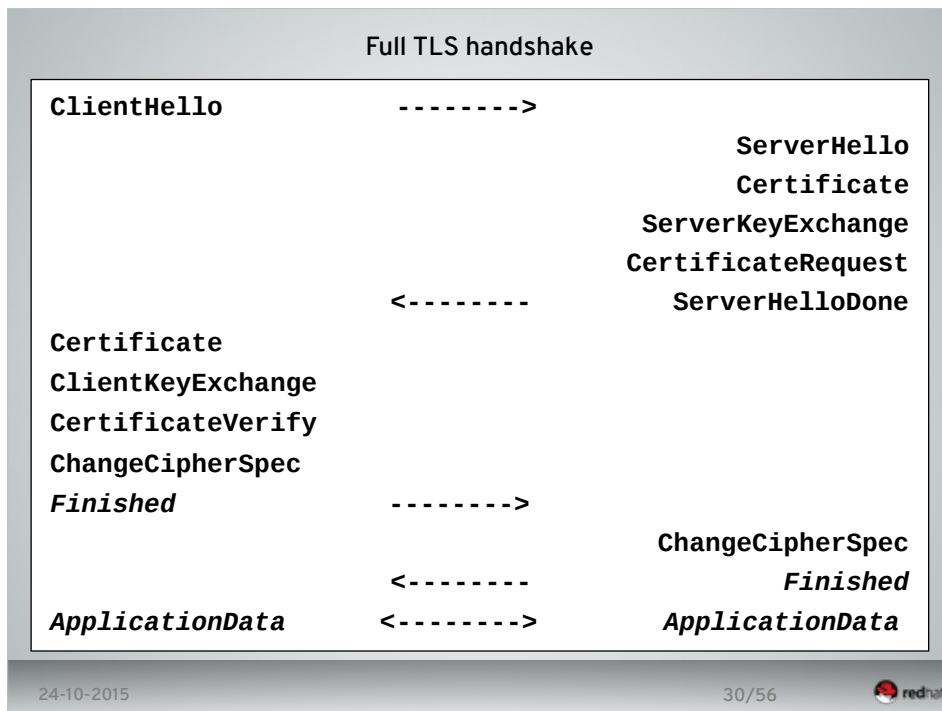
So to answer the “Why?” question: combination of understaffed projects, hard problem and resistance to change.



How can we fix this problem?

Duplication of effort

Of course, I didn't want to take every library in turn and extend its test suite. All of the libraries implement the same protocol so it should be possible to use one tool to test all of them. Even if not all features are implemented by all of them the protocol provides mechanisms for autodetection of features.



The problem with TLS is that's it's a complex protocol.

Some of those messages can easily have 20-30 fields and values that need to be set for negotiation to succeed, then we have values which are encrypted – we need to be able to modify the values before encryption.

Some of those messages are mandatory, others are optional.

When we know that the server actually accepted our connection? When we can decrypt the Finished message sent by server (that's the second line from bottom). In some cases we may actually be required to do a renegotiation (which is another just as long list) to reproduce an issue.

Existing fuzzers

24-10-2015

31/56



That makes it hard to use with existing frameworks.

I've looked at existing network protocol fuzzers. While there are few good protocol fuzzers, they focus more on protocols which don't have state (like HTTP) or have very limited state (like SMTP). This makes it hard to write fuzzers for TLS where a proper implementation needs to reference messages sent before and keep extensive state to be able to correctly encrypt and decrypt messages (including state between different connections – as that's needed for session resumption).

(Sulley and scapy.)

TLS testing (and fuzzing)

24-10-2015

32/56



So I decided that since I'll be mostly focusing on correctness of implementation and not on crashing the servers, starting work on a new test framework dedicated for testing TLS will be most effective. That being said, to make it as portable, easy to work with and extend as possible, I've decided to use pure Python implementation for both the framework as well as the underlying cryptography.

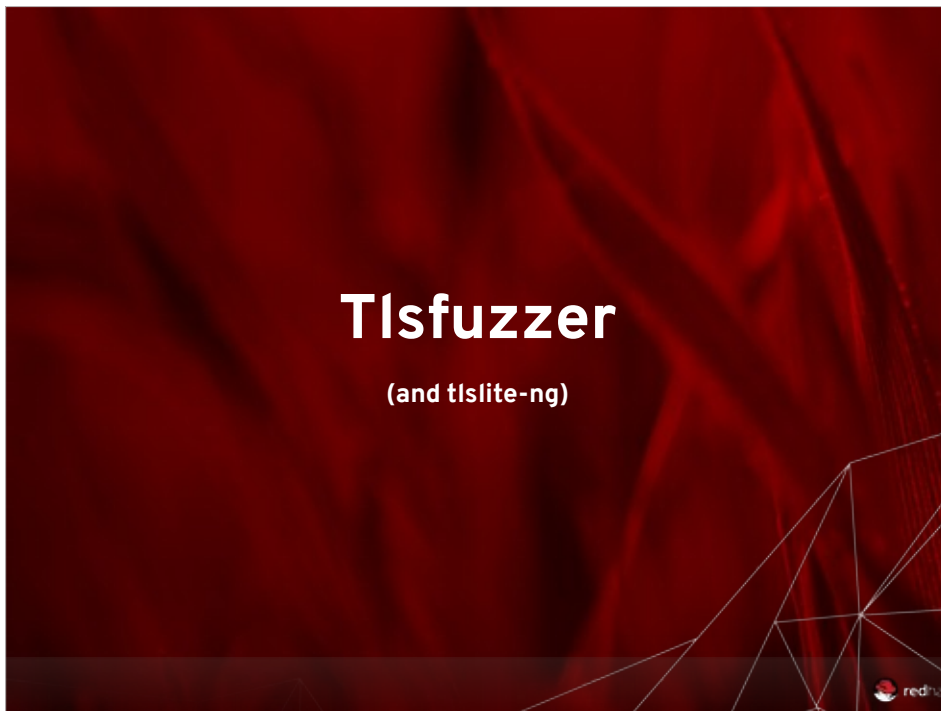
Timing information

24-10-2015

33/56



And last but not least, we need to be able to collect timing information for the handshakes and data to detect silent errors.



As such, I've started working on a tool that would enable testing, verification and in the end - fuzzing of any part of the TLS protocol.

Use cases

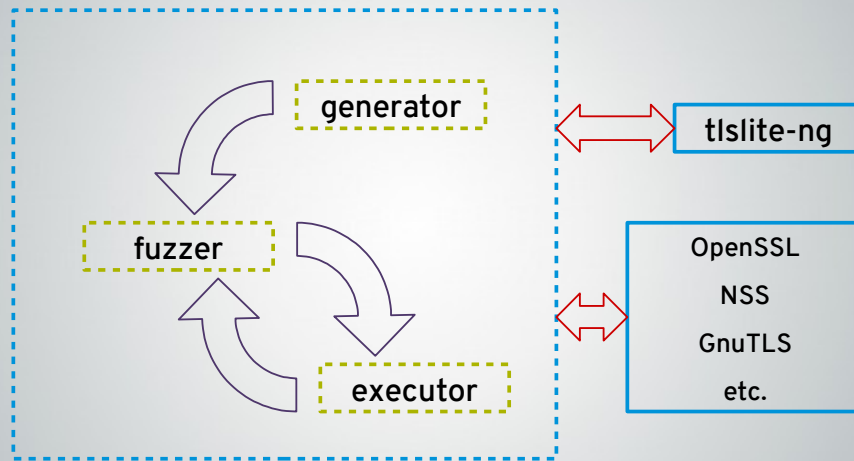
1. Manual run (setup)

2. Automated run

The manual run tells the user how to configure the server under test, detects the features supported by the server and then tests those features. If some features are mutually exclusive (like testing behaviour with and without client certificates), then it will ask the user to reconfigure the server in different way – in general, keeping track which features were already tested and which were not.

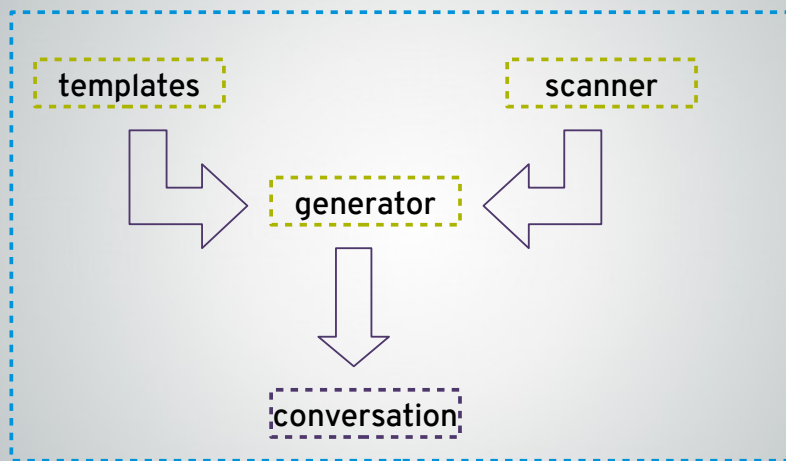
This also provides basic configuration for the fully automated run that later can be used inside system test suite so that it is run in continuous integration system.

Architecture



The process starts with the generator which takes connection templates and the knowledge about peer (what features it supports) and creates a set of message types and expected responses – a conversation. That is then modified by the fuzzer that knows which parts can be modified at will while modification of which will cause connection abort. Such modified message generators are passed to executor which connects to other framework using `tlslite-ng` implementation to create, write and encrypt the messages. If the connection aborts, the modified conversation can be saved for reproducing and fuzzer continues work.

Generator architecture



The generator probes the server to see which features it supports, takes the conversation templates that are applicable to it outputs conversations with expected responses applicable to a given server.

Fuzzing

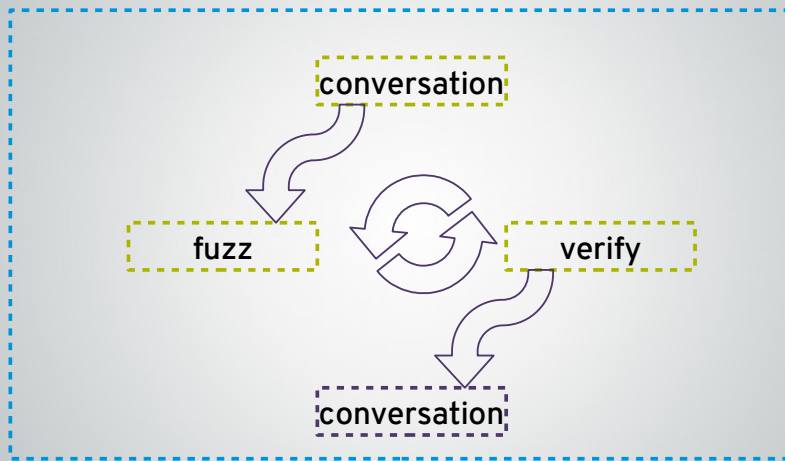
24-10-2015

38/56



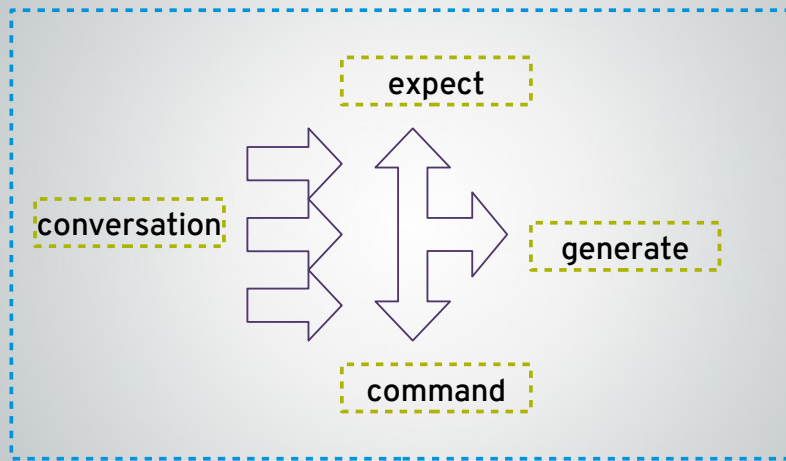
The important part here, is that while I'm talking about fuzzing it's not the typical fuzzing. Because the fuzzer is a TLS protocol fuzzer, the elements being fuzzed aren't single bytes, but rather whole messages or fields. The fuzzer knows when it inserts new messages, drops existing messages, fields, etc.

Fuzzer architecture



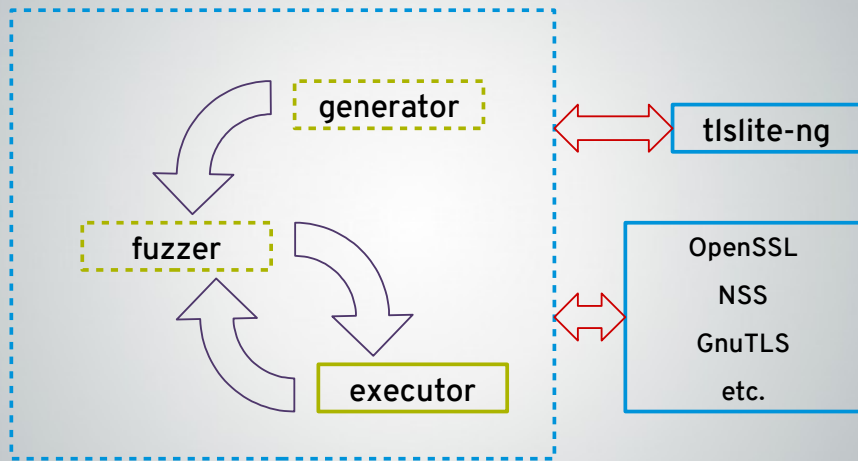
Fuzzer takes the conversation from the generator, modifies it randomly, taking into account that some changes will cause the expected responses to change too. So before it runs the conversation, it checks if it is self-consistent (that if the connection is supposed to be aborted, there's a check for that, and if it should succeed, there's check for that too).

Runner architecture



The runner then just takes decision nodes of the conversation and either expects a server response, generates a message or executes a generic command.

Architecture



At least, that's the plan. For now only the executor is working and I'm collecting interesting test cases to have better picture before starting to work on fuzzer. So what can we do with just the executor, turns out quite a bit.

Correct run

```
$ openssl s_server -key /tmp/localhost.key -cert /tmp/localhost.crt  
-www >/dev/null 2>&1  
$ PYTHONPATH=. python scripts/test-interleaved-application-data-and-  
fragmented-handshakes-in-renegotiation.py  
Application data inside Finished...  
OK  
Application data inside Client Key Exchange...  
OK  
Application data inside Client Hello...  
OK  
Test end  
successful: 3  
failed: 0
```

This is example of a run, showing a passing test.

Now, if we run one of the test cases I prepared against a correct implementation, we'll see just listing of specific test cases that passed and a summary

Failing run

```
$ openssl s_server -key /tmp/localhost.key -cert /tmp/localhost.crt  
-www >/dev/null 2>&1  
$ PYTHONPATH=. python scripts/test-interleaved-application-data-and-  
fragmented-handshakes-in-renegotiation.py  
(...snip...)  
Application data inside Client Hello...  
Error encountered while processing node  
<tlsfuzzer.expect.ExpectServerHello object at 0x7f0ac61d3310> with  
last message being: <tlslite.messages.Message object at  
0x7f0ac5f36a50>  
  
(...snip...)  
AssertionError: Unexpected message from peer: Alert(fatal,  
unexpected_message)  
  
Test end  
successful: 1  
failed: 2
```

If we run the same test case against a non-conformant server we will see the name of the test case that failed

Example test case

```
conversation = Connect("localhost", 4433)
node = conversation
ciphers = [CipherSuite.TLS_RSA_WITH_AES_128_CBC_SHA]
node = node.add_child(ClientHelloGenerator(ciphers))
node = node.add_child(ExpectServerHello())
node = node.add_child(ExpectCertificate())
node = node.add_child(ExpectServerHelloDone())
node = node.add_child(ClientKeyExchangeGenerator())
node = node.add_child(ChangeCipherSpecGenerator())
node = node.add_child(FinishedGenerator())
node = node.add_child(ExpectChangeCipherSpec())
node = node.add_child(ExpectFinished())
node = node.add_child(ApplicationDataGenerator(
    bytearray(b"hello server!\n")))
node = node.add_child(AlertGenerator(
    AlertLevel.warning,
    AlertDescription.close_notify))
node = node.add_child(ExpectAlert())
node.next_sibling = ExpectClose()
```

Under the hood it looks like this

But it's mostly just verbiage to create a decision tree

Example test case

```
conversation = Connect("localhost", 4433)
node = conversation
ciphers = [CipherSuite.TLS_RSA_WITH_AES_128_CBC_SHA]
node = node.add_child(ClientHelloGenerator(ciphers))
node = node.add_child(ExpectServerHello())
node = node.add_child(ExpectCertificate())
node = node.add_child(ExpectServerHelloDone())
node = node.add_child(ClientKeyExchangeGenerator())
node = node.add_child(ChangeCipherSpecGenerator())
node = node.add_child(FinishedGenerator())
node = node.add_child(ExpectChangeCipherSpec())
node = node.add_child(ExpectFinished())
node = node.add_child(ApplicationDataGenerator(
    bytearray(b"hello server!\n")))
node = node.add_child(AlertGenerator(
    AlertLevel.warning,
    AlertDescription.close_notify))
node = node.add_child(ExpectAlert())
node.next_sibling = ExpectClose()
```

24-10-2015

45/56

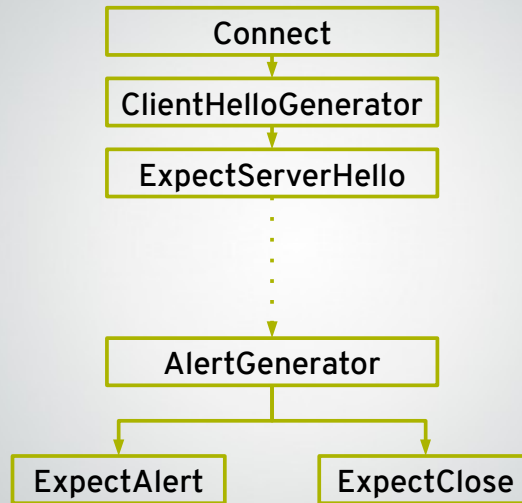


Under the hood it looks like this

The important stuff is in green here. Those are the tree nodes that are linked together, children are the “what to do if processing is successful” while siblings (see very bottom) are the “what alternatives do I have if the expectations are not met”.

The nodes in the tree all have sensible defaults (that is, behave according to the RFC and previously exchanged messages), with the exception of Connect node, which need to have host name and port provided.

Decision tree



That code creates a tree like this.

Generator then takes every node in turn and either tries to read a message from the record layer or write a message. If the read message doesn't match the expected one, or the server closes a connection, we get a connection abort.

Invalid extension test case

```
conversation = Connect("localhost", 4433)
node = conversation
ciphers = [CipherSuite.TLS_RSA_WITH_AES_128_CBC_SHA]
ext = {0 : # server_name extension ID
        lambda _: TLSExtension().create(0, bytearray(b'\xff'*4))}
node = node.add_child(ClientHelloGenerator(ciphers, extensions=ext))
node = node.add_child(ExpectAlert(AlertLevel.fatal,
                                   AlertDescription.decode_error))

alert_node = node
node = node.add_child(ExpectClose())

alert_node.next_sibling = ExpectClose()
```


While if we want to generate Client Hello message with invalid extension, we can simply specify the ID of the extension, its payload and link it to the Client Hello (yes, I know that I'm duplicating the info here, I'm still tweaking this part).

Since four bytes with all bits set is incorrect encoding of server_name extension, we can expect the server to send us a specific Alert (fatal, decode_error) or close the connection.

Handshake message format				
	Byte + 0	Byte + 1	Byte + 3	Byte + 4
Bytes 0..4	Message type	Message length		
Bytes 5..8	Version		Random (32 bytes)	
...	Session_ID length	Session_ID (0-32 bytes)		

24-10-2015

48/56



If we take a look at a Handshake protocol message format, there are 4 bytes that are common to all of the messages – the Message type, and the 3 byte length field (rest is beginning of a Client Hello)

Now, the protocol specifies, that the message length needs to match exactly the data encoded, in the given message.

But many fields in the messages have lengths of their own. So if we take a look here, the session_ID itself may specify a longer length (it's a byte, so up to 255) than the message length specifies.

Truncated message test case

```
conversation = Connect("localhost", 4433)
node = conversation
ciphers = [CipherSuite.TLS_RSA_WITH_AES_128_CBC_SHA]
node = node.add_child(truncate_handshake(
    ClientHelloGenerator(ciphers),
    1))
node = node.add_child(ExpectAlert(AlertLevel.fatal,
    AlertDescription.decode_error))

alert_node = node
node = node.add_child(ExpectClose())

alert_node.next_sibling = ExpectClose()
```

So, how do we test if the server handles truncated messages correctly: simple, just wrap message generator in `truncate_handshake` function and it will do the rest.

Padded message test case

```
conversation = Connect("localhost", 4433)
node = conversation
ciphers = [CipherSuite.TLS_RSA_WITH_AES_128_CBC_SHA]
node = node.add_child(pad_handshake(ClientHelloGenerator(ciphers),
    pad=bytearray(b'\xff\xff')))
node = node.add_child(ExpectAlert(AlertLevel.fatal,
    AlertDescription.decode_error))

alert_node = node
node = node.add_child(ExpectClose())

alert_node.next_sibling = ExpectClose()
```

Similarly for adding extra data at the end of a message.

There are also similar functions to modify arbitrary bytes of messages after they are created (especially useful for Finished messages). In similar simple way, it is possible to change behaviour of MAC or record padding for sending single message.

Features

- SSLv3, TLSv1.0, TLSv1.1 and TLSv1.2
- AES-CBC, AES-GCM, 3DES, RC4 and NULL ciphers
- MD5, SHA1, SHA256 and SHA384 HMAC
- RSA, SRP, SRP_RSA, DHE and DH_anon key exchange
- Encrypt-then-MAC
- TACK certificate pinning
- Client certificates
- Secure renegotiation
- TLS_FALLBACK_SCSV
- Next Protocol Negotiation
- ChaCha20/Poly1305 (soon™)
- ECDHE (soon™)

So the library currently supports most of the modern crypto, and some more obscure stuff. ECDHE, still needs some work in form of automated test suite, but it already is interoperable, working code. The ChaCha20 support is in even better shape, missing just last round of code review.

Missing stuff

- Drafts of TLSv1.3
- Extended master secret
- PSK key exchange
- ALPN
- AES-CCM
- CAMELLIA (CBC and GCM)
- ECDSA, DSA certificates
- Drafts of Curve25519
- Raw keys, GPG keys
- Heartbeat protocol
- Kerberos

The most obvious things missing is the support for extended master secret, pre-shared keys and ECDSA certificates. For testing Internet of Things gadgets we would need AES-CCM support.

Missing stuff

- **Test cases!**

And of course I'm missing test cases (future templates). So if you have ideas for tests (the more crazy the better), definitely please contact me.

Results

I haven't worked much on actual test cases, as for I've been focusing more on features needed to write those test cases, I still managed to find 9 bugs and create a simple reproducer for one complex issue in all three big TLS libraries: OpenSSL, NSS and GnuTLS. Just in a week's time. Ironically, the OpenSSL developers were the most responsive of them all.

Contributing

- <https://github.com/tomato42/tlsfuzzer>
- <https://github.com/tomato42/tlslite-ng>
- GPLv2 for tlsfuzzer
- LGPLv2 for tlslite-ng
- Tags **review request** and **help wanted**

The projects are on github, most of the code review is automated through Travis, Landscape.io and configuration of pylint.

I'm also using the review request and help wanted tags to signify places where you can jump in to help – and even questions like “why this code is like this” in code review are welcome, as that means that either the code is too complex or has not enough comments, so don't hold back because you think that you won't be able to contribute because you don't know the project.

Questions?

Contact: hkario@redhat.com

Project: <https://github.com/tomato42/tlsfuzzer>



Questions?