TLS Test Framework

Hubert Kario
Senior Quality Engineer
05-02-2017

SSL is dead! Long live ~~SSL~~ TLS!

What is TLS?

- Transport Layer Security
- Protocol used to provide security for HTTP**S**
- Over 50% of Internet web traffic is using TLS
- E-mail, Tor, IoT, SRTP, VPN ...

Why use TLS?

- Confidentiality
- Integrity
- Authenticity

History of TLS

- SSL version 1 – Netscape – 1990's (internal)
- SSL version 2 – Netscape – 1995
- SSL version 3 – Netscape – 1996
- TLS version 1.0 – IETF – 1999
- TLS version 1.1 – IETF – 2006
- TLS version 1.2 – IETF – 2008
- TLS version 1.3 (draft 18) – IETF – 2016

Updates to TLS

- Nearly 70 RFCs
- Around 6 widely deployed drafts
- 27 TLS extensions
- Fixes for protocol and implementation bugs:
  - Secure renegotiation
  - fallback_scsv
  - Client Hello padding
  - Extended master secret (Triple handshake)

Complexity of TLS

- 326 ciphersuites (official)
- 15 key sizes and types (common)
- 4 PKI cryptosystems
- 37 Diffie-Hellman group sizes (common and defined)
- 16 signature-hash algorithm pairs
- 4 extensions directly interacting with derivation and use of cryptographic keys
- 4 modes of connection establishment
- Client certificates

Testing with TLS libraries

Existing network protocol fuzzers

tlsfuzzer

tlsfuzzer

- Python 2.6 or 3.2 and up
- No native dependencies (pure Python)
- Extensive test suite over 60 test scripts/features
- Over 18000 individual test cases
- Relatively fast: ~300 tc/s/core on 2.6Ghz Skylake i7
- Focus on new crypto and TLS features
- Unit tested: 95.5% statement coverage
- Over 20 issues found across NSS, GnuTLS and OpenSSL

“fuzzer”?

Why use tlsfuzzer?

- Configuration specifics
- Integration testing
- Black box testing
- Forward compatibility
- Obscure client compatibility

Simple compatibility test

```
conversation = Connect(host, port)
node = conversation
ciphers = [CipherSuite.TLS_EMPTY_RENEGOTIATION_INFO_SCSV,
           CipherSuite.TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA384,
           ...,
           CipherSuite.TLS_RSA_WITH_RC4_128_MD5]
ext = {}
ext[ExtensionType.server_name] = SNIExtension().create(b"example.io")
ext[ExtensionType.supported_groups] = SupportedGroupsExtension().\
    create([23, 24, 25])
ext[ExtensionType.signature_algorithms] = \
    SignatureAlgorithmsExtension().create([
        (HashAlgorithm.sha384, SignatureAlgorithm.rsa),
        (HashAlgorithm.sha256, SignatureAlgorithm.rsa),
        (HashAlgorithm.sha1, SignatureAlgorithm.rsa),
        (HashAlgorithm.sha256, SignatureAlgorithm.ecdsa),
        (HashAlgorithm.sha1, SignatureAlgorithm.ecdsa)])
node = node.add_child(ClientHelloGenerator(ciphers, ext, (3, 3)))
node = node.add_child(ExpectServerHello())
```

Complete connection example

```
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(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( … ))
node = node.add_child(ExpectApplicationData())
node = node.add_child(AlertGenerator(AlertLevel.warning,
                                     AlertDescription.close_notify))
node = node.add_child(ExpectAlert(AlertLevel.warning,
                                  AlertDescription.close_notify))
node.next_sibling = ExpectClose()
node.add_child(ExpectClose())
```

CCS injection attack

```
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(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( … ))
node = node.add_child(ExpectApplicationData())
node = node.add_child(AlertGenerator(AlertLevel.warning,
                                     AlertDescription.close_notify))
node = node.add_child(ExpectAlert(AlertLevel.warning,
                                  AlertDescription.close_notify))
node.next_sibling = ExpectClose()
node.add_child(ExpectClose())
```

CCS injection attack

```
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(ExpectCertificate())
node = node.add_child(ExpectServerHelloDone())
node = node.add_child(ChangeCipherSpecGenerator())
node = node.add_child(ExpectAlert(AlertLevel.fatal,
                                  AlertDescription.unexpected_message))
node.add_child(ExpectClose())
```

Malformed message

```
conversation = Connect(host, port)
node = conversation
ciphers = [CipherSuite.TLS_RSA_WITH_AES_128_CBC_SHA,
           CipherSuite.TLS_EMPTY_RENEGOTIATION_INFO_SCSV]
ext = {ExtensionType.server_name :
       SNIExtension().create(b“example.io”)}
hello = ClientHelloGenerator(ciphers, ext)
node = node.add_child(truncate_handshake(hello, 4))
node = node.add_child(ExpectAlert(AlertLevel.fatal,
                                  AlertDescription.decode_error))
node.add_child(ExpectClose())
```

Script automation

```
[
    {"server_command": ["{command}", "s_server",
                        "-key", "tests/serverX509Key.pem",
                        "-cert", "tests/serverX509Cert.pem",
                        "-www",
                        "-accept", "4433"],
     "tests" : [
         {"name" : "test-aes-gcm-nonces.py" },
         {"name" : "test-atypical-padding.py" }
     ]
    }
]
```

Script automation

```
$ python tests/scripts_retention.py openssl-1.0.2.json `which openssl`
INFO:__main__:Server process started
server:stdout:Using default temp DH parameters
server:stdout:ACCEPT
INFO:__main__:test-aes-gcm-nonces.py:started
INFO:__main__:test-aes-gcm-nonces.py:finished
INFO:__main__:test-atypical-padding.py:started
INFO:__main__:test-atypical-padding.py:finished
DEBUG:root:Killing server process
DEBUG:root:Server process killed: -15
Ran 5 test cases
expected pass: 5
expected fail: 0

Ran 2 scripts
good: 2
bad: 0
```

TLS Features

- SSLv2, SSLv3, TLSv1.0, TLSv1.1 and TLSv1.2
- AES-CBC, AES-GCM, 3DES, RC4, Chacha20 and NULL ciphers
- MD5, SHA1, SHA256 and SHA384 HMAC
- RSA, SRP, SRP_RSA, DHE, DH_anon, ECDHE and ECDH_anon key exchange
- Encrypt-then-MAC
- TACK certificate pinning
- Client certificates
- Secure renegotiation
- TLS_FALLBACK_SCSV
- Extended master secret

TLS Features

- Next Protocol Negotiation
- Application Layer Protocol Negotiation
- FFDHE group negotiation (RFC 7919)
- Server name indication

Missing TLS features

- TLSv1.3 (draft) – in progress!
- RSA-PSS
- X25519
- PSK key exchange
- AES-CCM
- Camellia (CBC and GCM)
- ECDSA, DSA certificates
- Raw keys, GPG keys
- Hearbeat protocol
- Kerberos

Contributing

https://github.com/tomato42/tlsfuzzer

https://github.com/tomato42/tlslite-ng

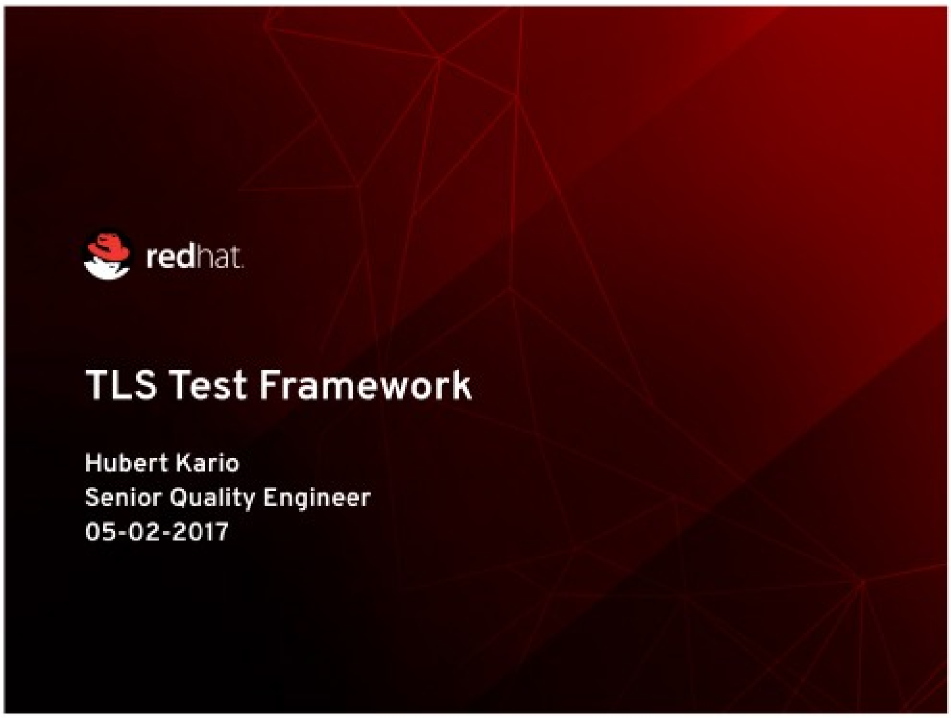
GPLv2 for tlsfuzzer

LGPLv2 for tlslite-ng

Tags review request and up for grabs

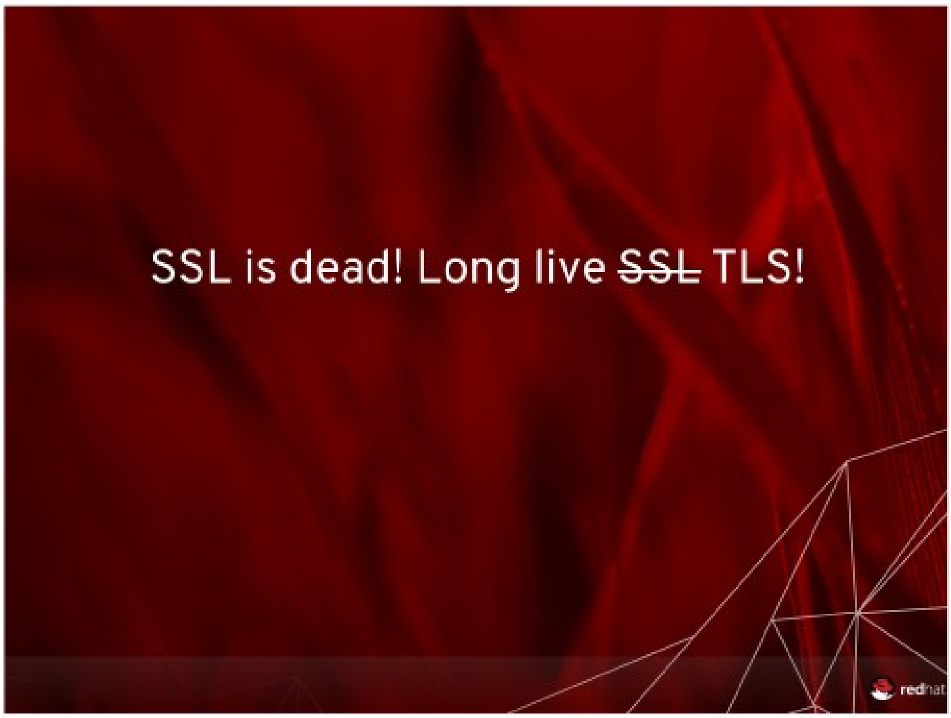
Questions?

hkario@redhat.com



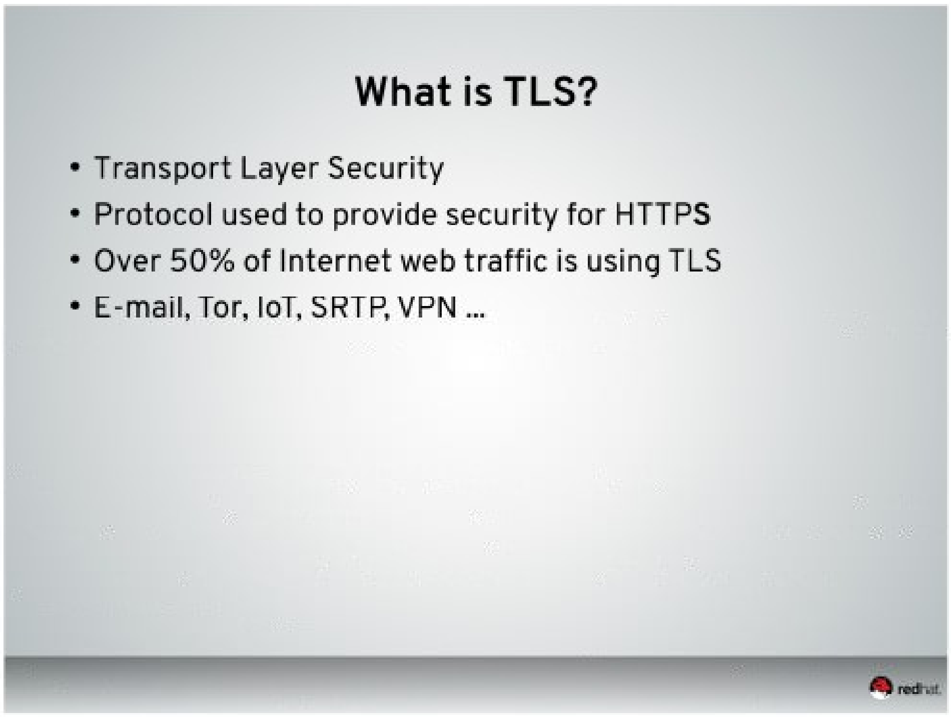
TLS Test Framework

Hubert Kario
Senior Quality Engineer
05-02-2017



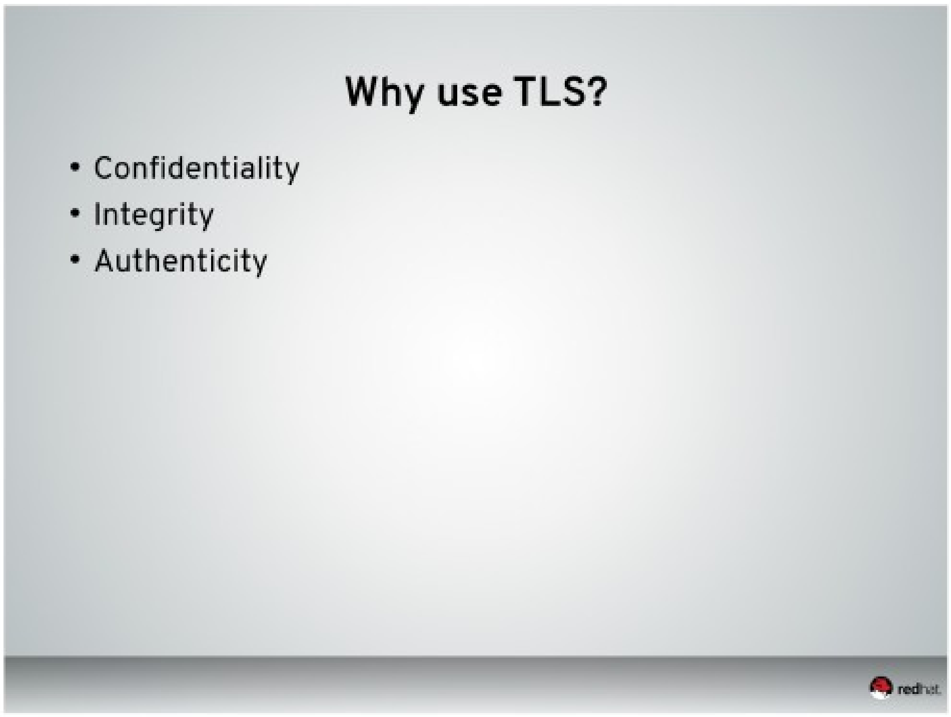
SSL is dead! Long live ~~SSL~~ TLS!

SSL is the name of the first two versions of what has later become the TLS protocol. Both of those versions – SSLv2 and SSLv3 – are now known to be insecure and they are officially deprecated. So for the rest of the presentation I will be using the acronym TLS.



What is TLS?

- Transport Layer Security
- Protocol used to provide security for HTTP**S**
- Over 50% of Internet web traffic is using TLS
- E-mail, Tor, IoT, SRTP, VPN …



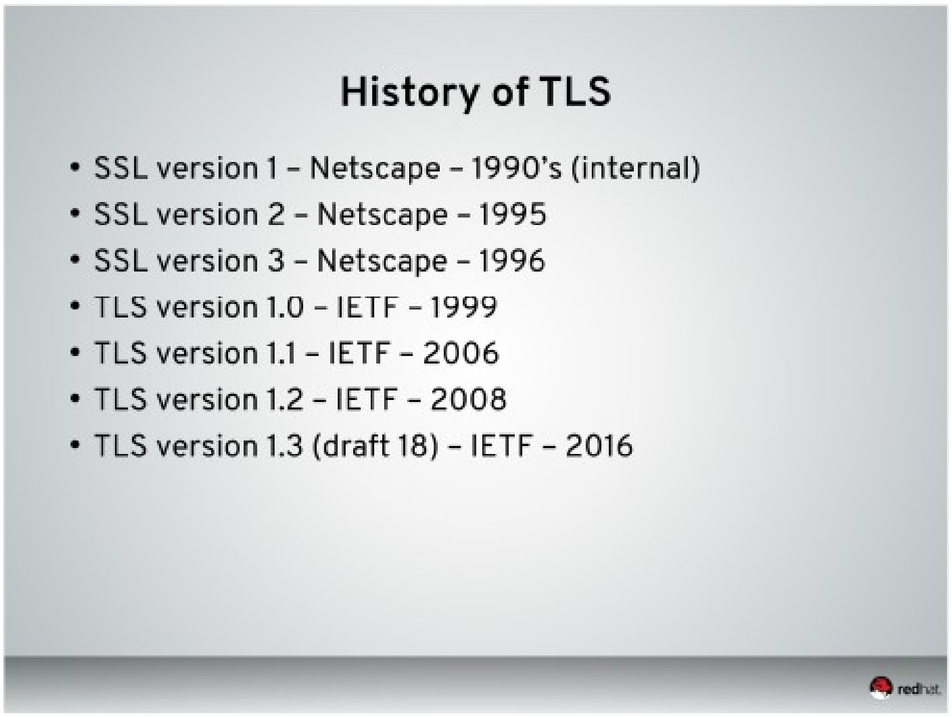
Why use TLS?

- Confidentiality
- Integrity
- Authenticity

Confidentiality is the property of the protocol that makes the contents of communication secret

Integrity establishes that no messages were modified in the transmission and that all come from the same server or client

Authenticity establishes the identity of the server you are communicating with



History of TLS

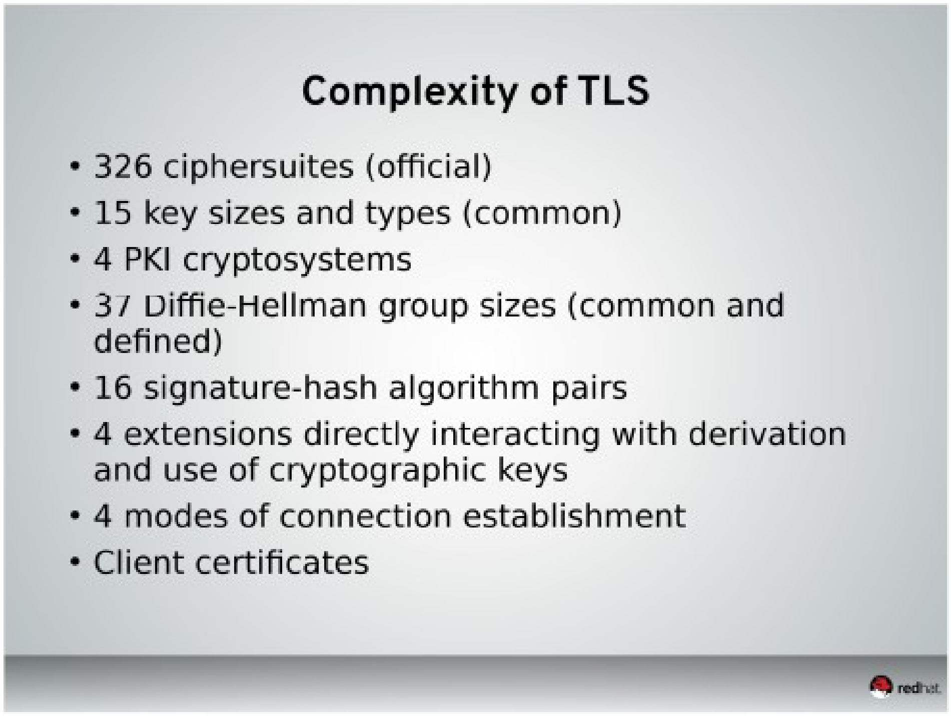
- SSL version 1 – Netscape – 1990's (internal)
- SSL version 2 – Netscape – 1995
- SSL version 3 – Netscape – 1996
- TLS version 1.0 – IETF – 1999
- TLS version 1.1 – IETF – 2006
- TLS version 1.2 – IETF – 2008
- TLS version 1.3 (draft 18) – IETF – 2016

IETF = Internet Engineering Task Force

In other words, the basics of the protocol were established 20 years ago now.

Updates to TLS

- Nearly 70 RFCs
- Around 6 widely deployed drafts
- 27 TLS extensions
- Fixes for protocol and implementation bugs:
  - Secure renegotiation
  - fallback_scsv
  - Client Hello padding
  - Extended master secret (Triple handshake)

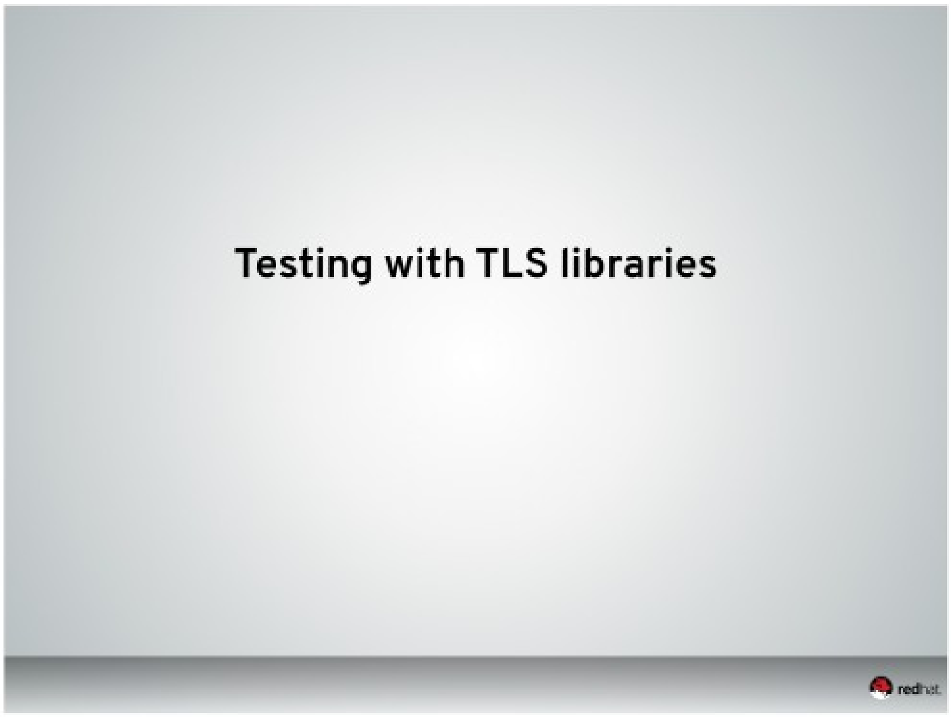


Complexity of TLS

- 326 ciphersuites (official)
- 15 key sizes and types (common)
- 4 PKI cryptosystems
- 37 Diffie-Hellman group sizes (common and defined)
- 16 signature-hash algorithm pairs
- 4 extensions directly interacting with derivation and use of cryptographic keys
- 4 modes of connection establishment
- Client certificates

Most of those things interacts with each-other, the order of items advertised by peers does matter.

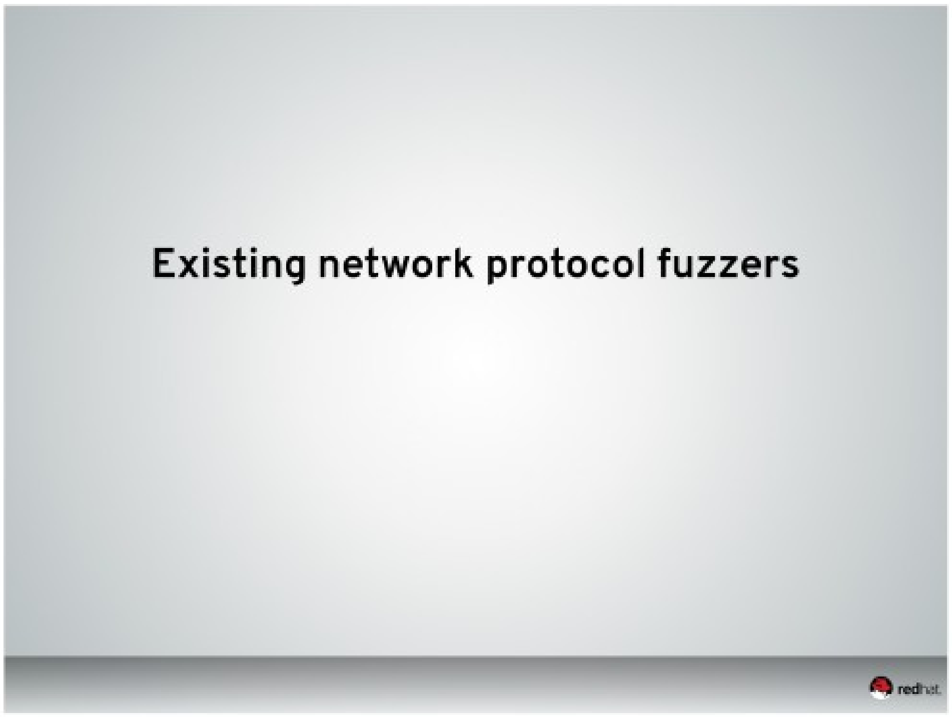
And not only they need to be correct, in that client or server computes valid ciphertext and signatures, it needs to be tested that it also verifies the validity of the signatures or correctness of encryption padding.



Testing with TLS libraries

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.

Moreover, they deprecate features like insecure protocols or algorithms, it's generally not possible to make them advertise features they don't support.



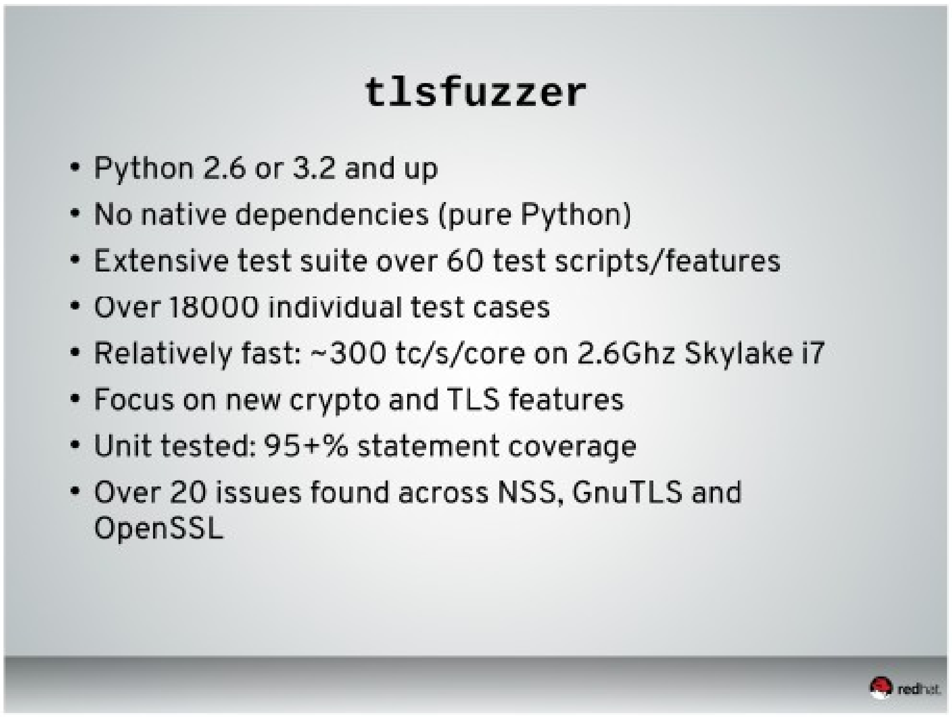
Existing network protocol fuzzers

As I've said, TLS is a complex protocol, if only because of all the accretion that happened over the years.

I've looked at existing network protocol fuzzers. While there 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).

tlsfuzzer

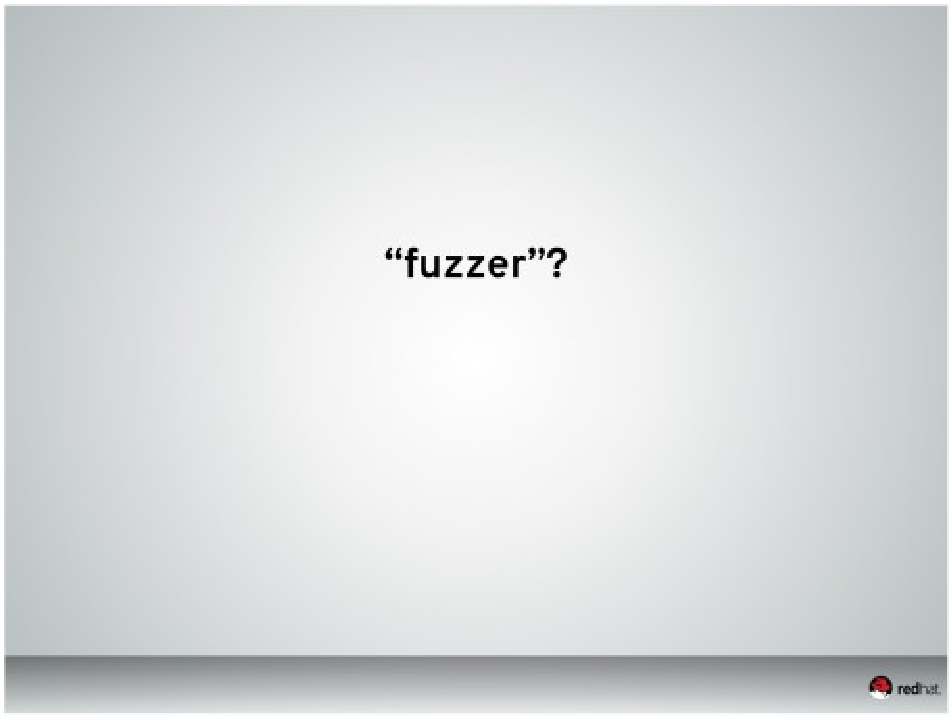
That's why we have started on a new dedicated TLS protocol test framework called tlsfuzzer.



tlsfuzzer

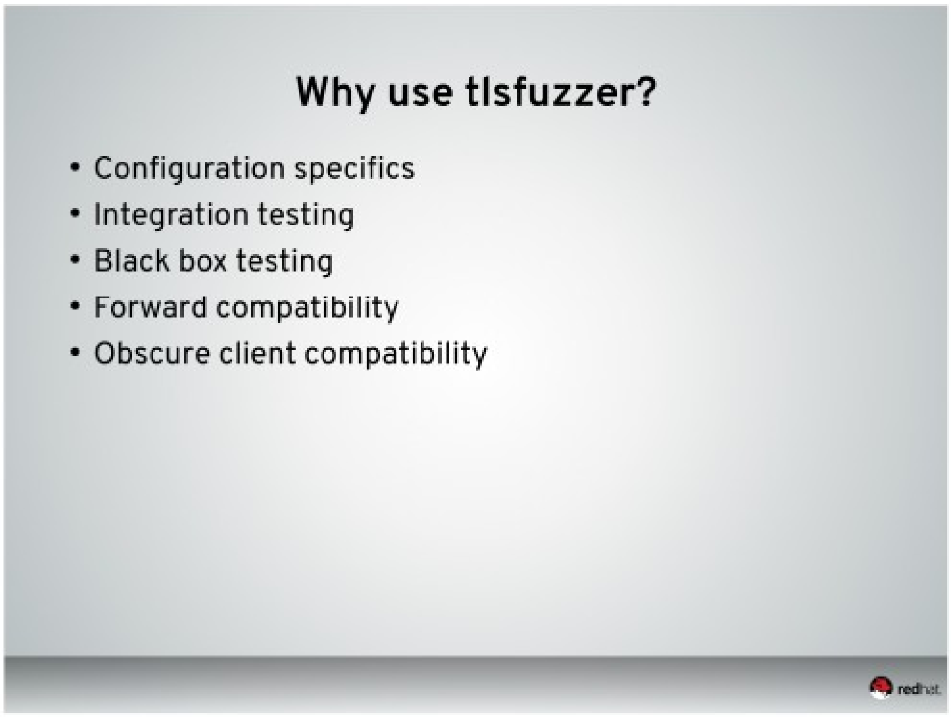
- Python 2.6 or 3.2 and up
- No native dependencies (pure Python)
- Extensive test suite over 60 test scripts/features
- Over 18000 individual test cases
- Relatively fast: ~300 tc/s/core on 2.6Ghz Skylake i7
- Focus on new crypto and TLS features
- Unit tested: 95+% statement coverage
- Over 20 issues found across NSS, GnuTLS and OpenSSL

We have found in total over 20 issues in NSS, GnuTLS and OpenSSL and one security issue in NSS.



“fuzzer”?

While I am talking about a fuzzer, it is a TLS protocol fuzzer, so it is focusing on TLS messages and fields contents, not doing random changes to the network packets sent.



Why use tlsfuzzer?

- Configuration specifics
- Integration testing
- Black box testing
- Forward compatibility
- Obscure client compatibility

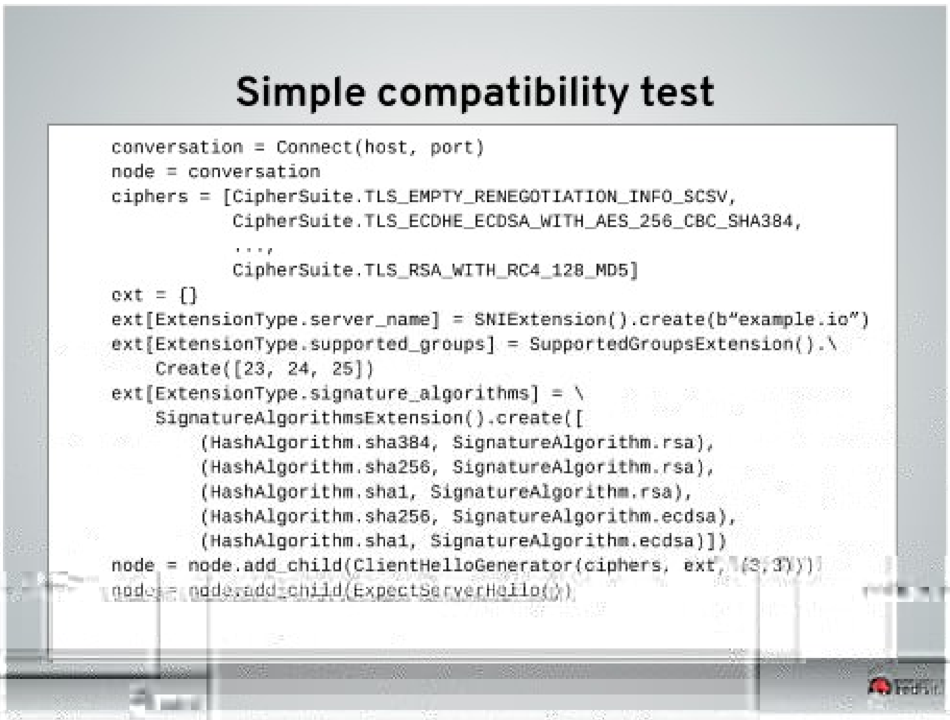
While usefulness of a protocol fuzzer is quite obvious to people writing TLS implementations, it is also useful for anyone using TLS libraries.

Not every single combination of configuration parameters is tested (or even can be tested) by the implementation programmers.

Some features need callbacks which are written by programmers of the application using the TLS implementation.

With appliances usually very little is known about the TLS implementation they are using, especially if it is not a dedicated TLS terminator but rather a UPS, firewall, spam filter, etc. and TLS is mainly used for administrative interface.

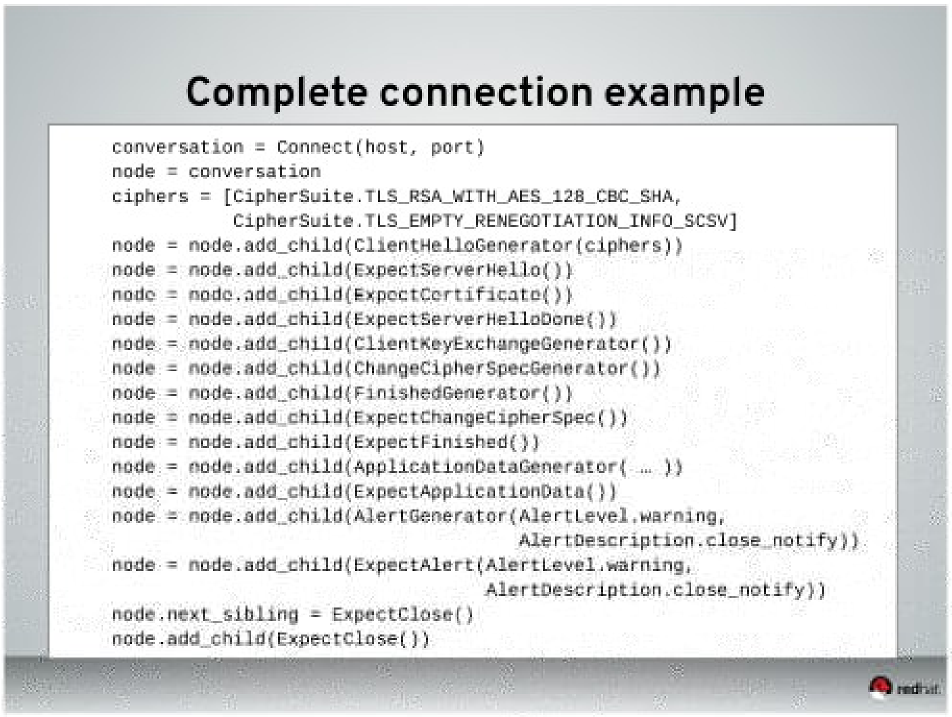
In general, this allows to test how any given implementation will be able to handle possible future version of protocol – as while the exact changes are unknown, the possible changes they can introduce are known – the initial Client Hello message sent will need to remain backwards compatible to currently deployed version of TLS, be it 1.2 or the upcoming 1.3.



Simple compatibility test

```
conversation = Connect(host, port)
node = conversation
ciphers = [CipherSuite.TLS_EMPTY_RENEGOTIATION_INFO_SCSV,
           CipherSuite.TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA384,
           ...,
           CipherSuite.TLS_RSA_WITH_RC4_128_MD5]
ext = {}
ext[ExtensionType.server_name] = SNIExtension().create(b"example.io")
ext[ExtensionType.supported_groups] = SupportedGroupsExtension().\
    Create([23, 24, 25])
ext[ExtensionType.signature_algorithms] = \
    SignatureAlgorithmsExtension().create([
        (HashAlgorithm.sha384, SignatureAlgorithm.rsa),
        (HashAlgorithm.sha256, SignatureAlgorithm.rsa),
        (HashAlgorithm.sha1, SignatureAlgorithm.rsa),
        (HashAlgorithm.sha256, SignatureAlgorithm.ecdsa),
        (HashAlgorithm.sha1, SignatureAlgorithm.ecdsa)])
node = node.add_child(ClientHelloGenerator(ciphers, ext, (3,3)))
node = node.add_child(ExpectServerHello())
```

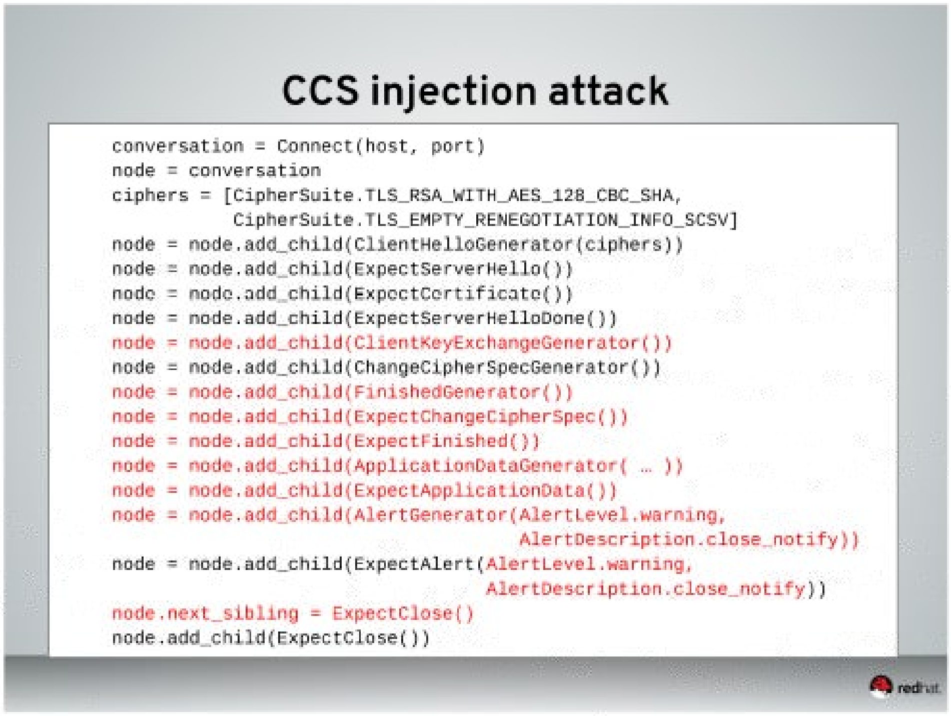
We can create client hello messages exactly like would a given client create, impersonating it and checking if the server will interoperate with it.



Complete connection example

```
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(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( … ))
node = node.add_child(ExpectApplicationData())
node = node.add_child(AlertGenerator(AlertLevel.warning,
                                     AlertDescription.close_notify))
node = node.add_child(ExpectAlert(AlertLevel.warning,
                                  AlertDescription.close_notify))
node.next_sibling = ExpectClose()
node.add_child(ExpectClose())
```

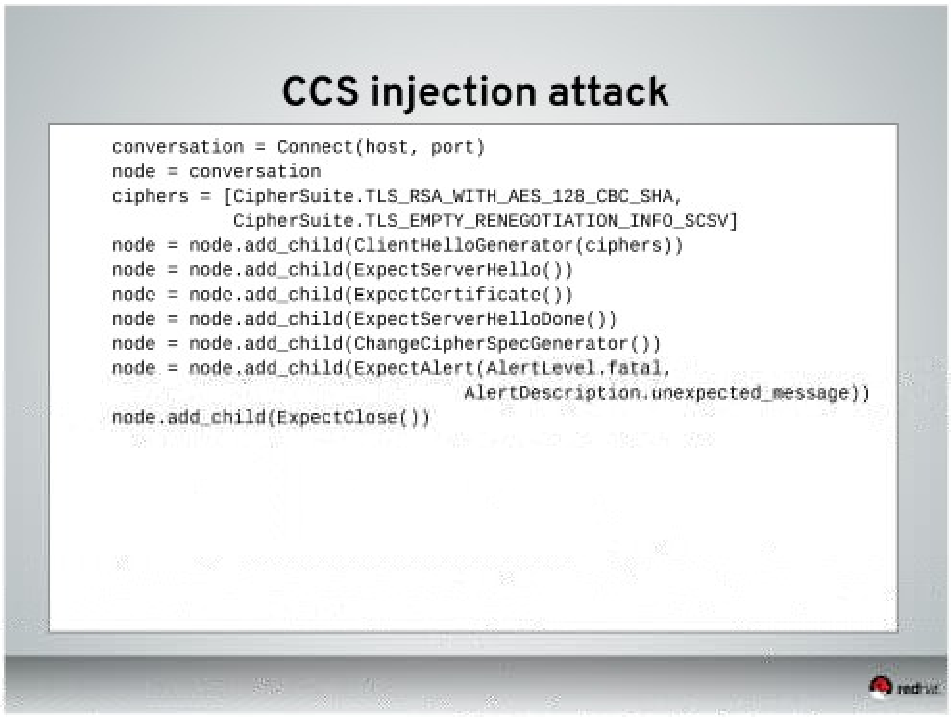
We can also perform a full connection, to verify if the server implements the algorithms correctly.



CCS injection attack

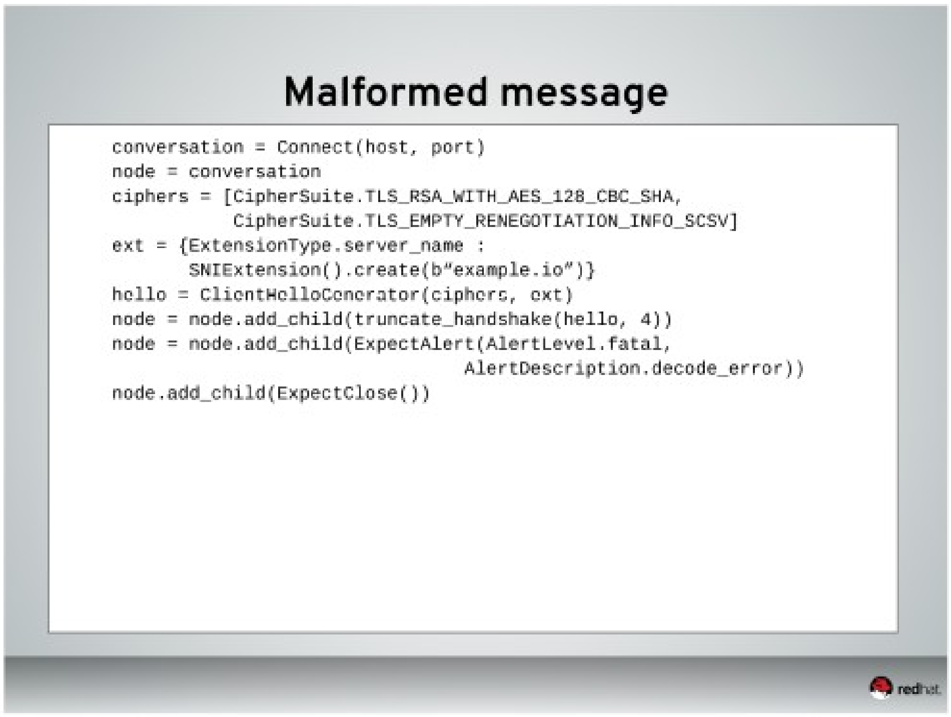
```
conversation = Connect(host, port)
node = conversation
ciphers = [CipherSuite.TLS_RSA_WITH_AES_128_CBC_SHA,
           CipherSuite.TLS_EMPTY_RENEGOTIATION_INFO_SCSV]
node = node.add_child(ClientHelloGenerator(ciphers))
node = node.add_child(ExpectServerHello())
node = node.add_child(ExpectCertificate())
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( … ))
node = node.add_child(ExpectApplicationData())
node = node.add_child(AlertGenerator(AlertLevel.warning,
                                     AlertDescription.close_notify))
node = node.add_child(ExpectAlert(AlertLevel.warning,
                                  AlertDescription.close_notify))
node.next_sibling = ExpectClose()
node.add_child(ExpectClose())
```

We can replicate attacks, like the Change Cipher Spec bug in OpenSSL where that message was accepted too early, making the encryption keys predictable. To create it, we need to remove most of the messages and place the CCS earlier.


CCS injection attack

```
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(ExpectCertificate())
node = node.add_child(ExpectServerHelloDone())
node = node.add_child(ChangeCipherSpecGenerator())
node = node.add_child(ExpectAlert(AlertLevel.fatal,
                                  AlertDescription.unexpected_message))
node.add_child(ExpectClose())
```

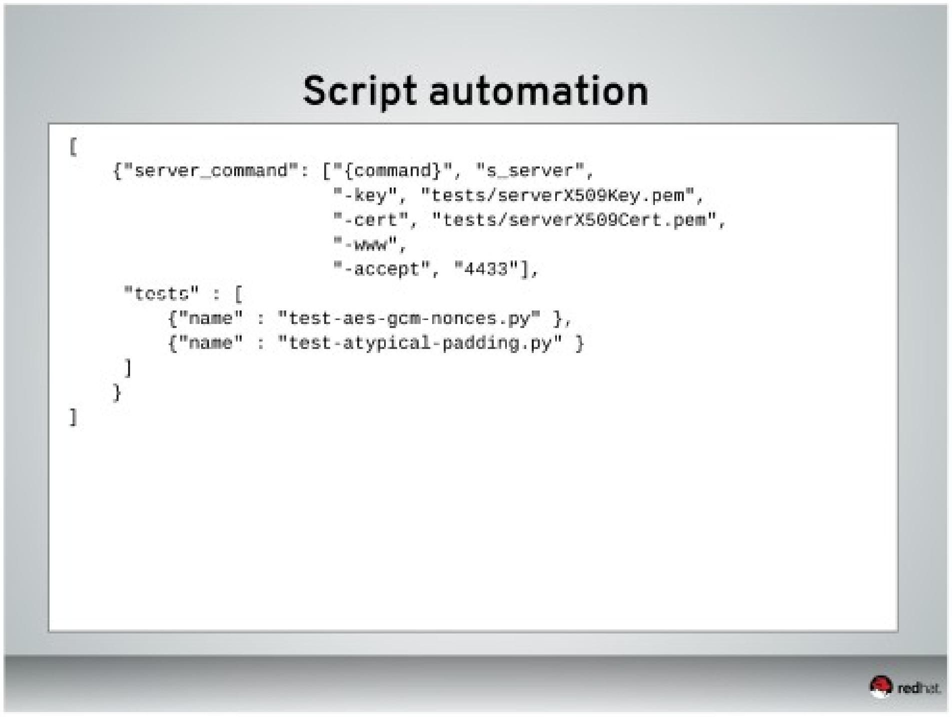
That change should make the server reply with unexpected_message alert message.



Malformed message

```
conversation = Connect(host, port)
node = conversation
ciphers = [CipherSuite.TLS_RSA_WITH_AES_128_CBC_SHA,
           CipherSuite.TLS_EMPTY_RENEGOTIATION_INFO_SCSV]
ext = {ExtensionType.server_name :
       SNIExtension().create(b"example.io")}
hello = ClientHelloGenerator(ciphers, ext)
node = node.add_child(truncate_handshake(hello, 4))
node = node.add_child(ExpectAlert(AlertLevel.fatal,
                                  AlertDescription.decode_error))
node.add_child(ExpectClose())
```

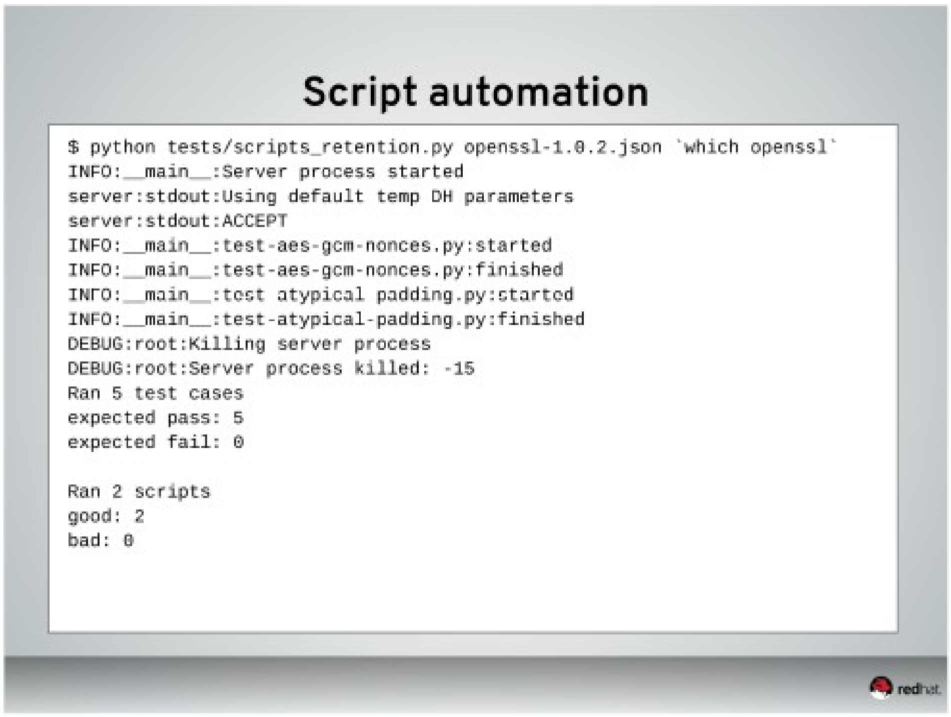
It’s also possible to create malformed messages, like in this case, a client hello that lacks the last 4 bytes of the message.



Script automation

```
[
    {"server_command": ["{command}", "s_server",
                        "-key", "tests/serverX509Key.pem",
                        "-cert", "tests/serverX509Cert.pem",
                        "-www",
                        "-accept", "4433"],
     "tests" : [
         {"name" : "test-aes-gcm-nonces.py" },
         {"name" : "test-atypical-padding.py" }
     ]
    }
]
```

I've also recently added support for automating of running of the scripts against a server. The server is automatically started and stopped by the server, and then multiple scripts are run against it. Both the server and scripts can receive optional parameters, or may be expected to fail (in case a test case is failing because of a missing feature in the SUT).

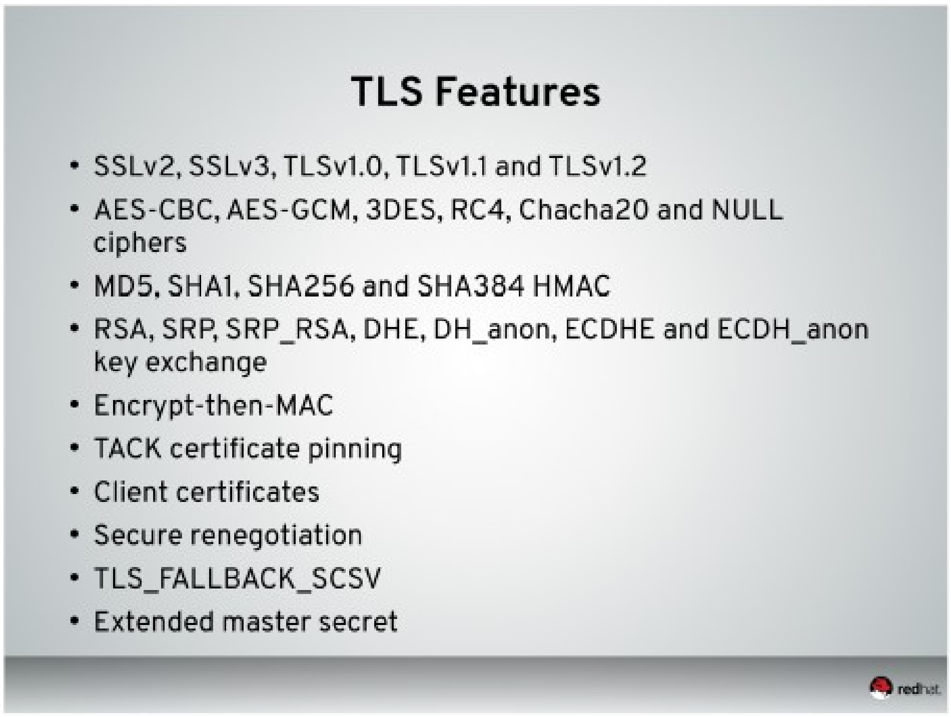


Script automation

```
$ python tests/scripts_retention.py openssl-1.0.2.json `which openssl`
INFO:__main__:Server process started
server:stdout:Using default temp DH parameters
server:stdout:ACCEPT
INFO:__main__:test-aes-gcm-nonces.py:started
INFO:__main__:test-aes-gcm-nonces.py:finished
INFO:__main__:test-atypical-padding.py:started
INFO:__main__:test-atypical-padding.py:finished
DEBUG:root:Killing server process
DEBUG:root:Server process killed: -15
Ran 5 test cases
expected pass: 5
expected fail: 0

Ran 2 scripts
good: 2
bad: 0
```

Running the previous script will give us such summary.

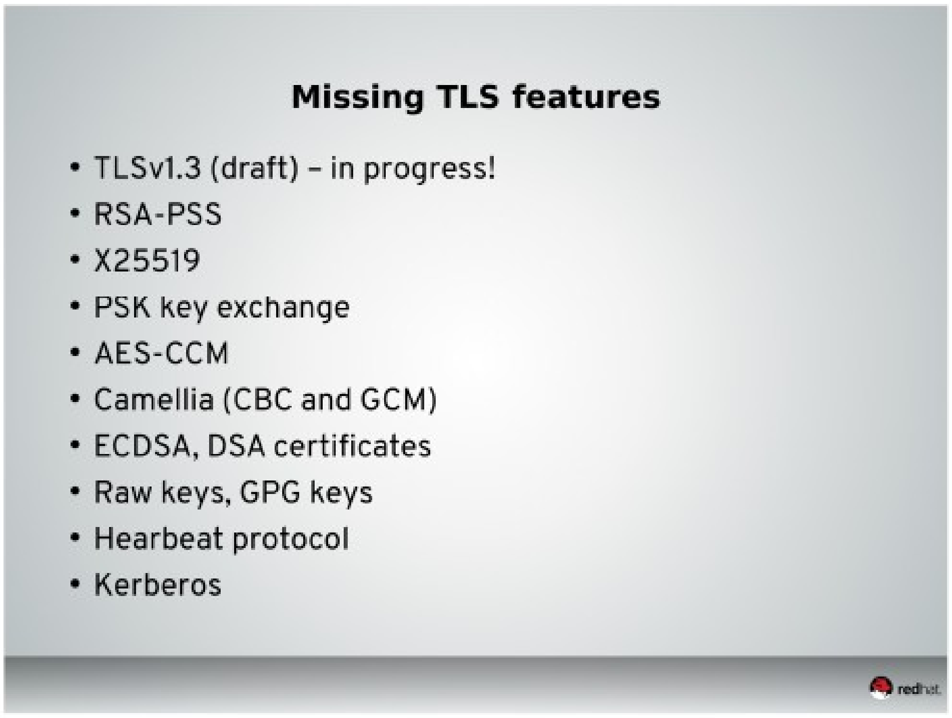


TLS Features

- SSLv2, SSLv3, TLSv1.0, TLSv1.1 and TLSv1.2
- AES-CBC, AES-GCM, 3DES, RC4, Chacha20 and NULL ciphers
- MD5, SHA1, SHA256 and SHA384 HMAC
- RSA, SRP, SRP_RSA, DHE, DH_anon, ECDHE and ECDH_anon key exchange
- Encrypt-then-MAC
- TACK certificate pinning
- Client certificates
- Secure renegotiation
- TLS_FALLBACK_SCSV
- Extended master secret

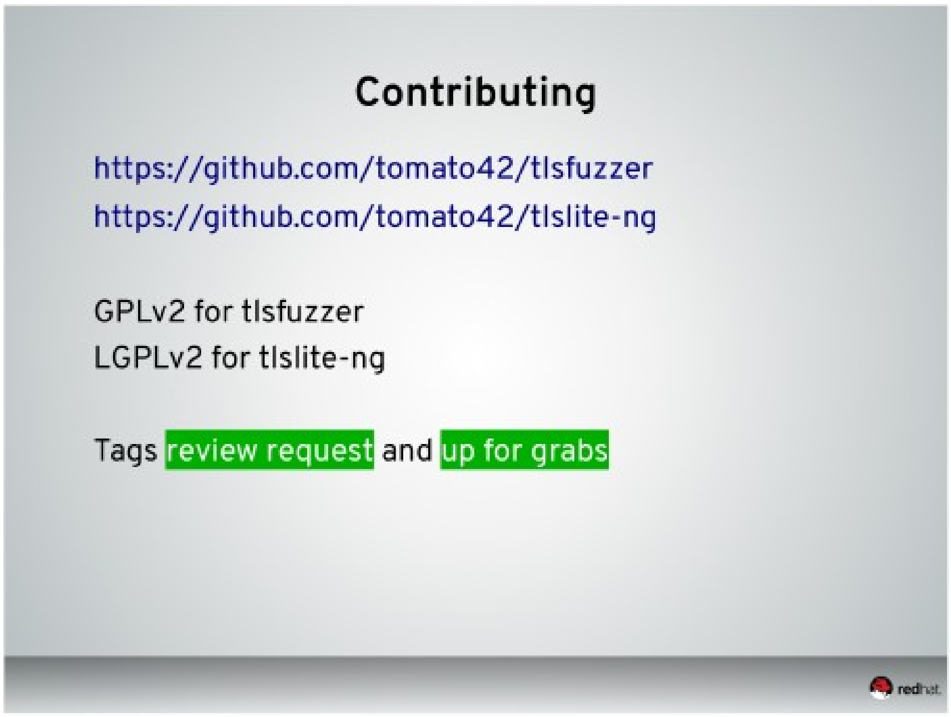
TLS Features

- Next Protocol Negotiation
- Application Layer Protocol Negotiation
- FFDHE group negotiation (RFC 7919)
- Server name indication



Missing TLS features

- TLSv1.3 (draft) – in progress!
- RSA-PSS
- X25519
- PSK key exchange
- AES-CCM
- Camellia (CBC and GCM)
- ECDSA, DSA certificates
- Raw keys, GPG keys
- Hearbeat protocol
- Kerberos



Contributing

https://github.com/tomato42/tlsfuzzer
https://github.com/tomato42/tlslite-ng

GPLv2 for tlsfuzzer
LGPLv2 for tlslite-ng

Tags review request and up for grabs

Questions?

hkario@redhat.com