rustls

a modern tls stack in rust

why

wanted to learn rust by doing

dissatisfaction with 'toolkit' approach

Brian Smith's webpki and *ring* crates are significant building blocks

(but nobody had made a full TLS stack using them)

approx my 4th TLS stack (others proprietary or pentest-focussed)

(so I thought I might do a reasonable job this time around)

approach

make something that is easy and **safe** for 90% of uses

- but perhaps might not ever cater to the remaining 10%

"if in doubt, leave it out"

don't require configuration for 90% of such uses

- sane, well-regarded defaults

"The most dangerous code in the world: validating SSL certificates in non-browser software" - Boneh et al, ACM CCS'12

"Our main conclusion is that *SSL certificate validation is completely broken in many critical software applications and libraries*" ...

"For the most part, the actual SSL libraries used in these programs are correct" ...

"The primary cause of these vulnerabilities is the developers' misunderstanding of the numerous options, parameters, and return values of SSL libraries" ...

"many SSL libraries are unsafe by default, requiring higher-level software to correctly set their options, provide hostname verification functions [...]"

"APIs should present high-level abstractions to developers, such as "confidential and authenticated tunnel," as opposed to requiring them to explicitly deal with low-level details such as hostname verification"

https://www.cs.utexas.edu/~shmat/shmat_ccs12.pdf

current TLS feature list

TLS1.2 and TLS1.3-draft-18

Client and server end

AES-128-GCM, AES-256-GCM, chacha20poly1305 suites

Forward secrecy, always

Server authentication, always

Optional client authentication

All kinds of resumption

current TLS non-feature list

SSL2, SSL3, TLS1.0, TLS1.1

Renegotiation (it's nasty, broken several times, killed in TLS1.3)

RC4, DES, Triple-DES, discrete log DH, EXPORT suites, …

Suites not providing forward secrecy

No support for TLS1.3 0RTT data (scary!)

(maybe one day… but needs extreme care)

about TLS1.0/1.1

hard decision

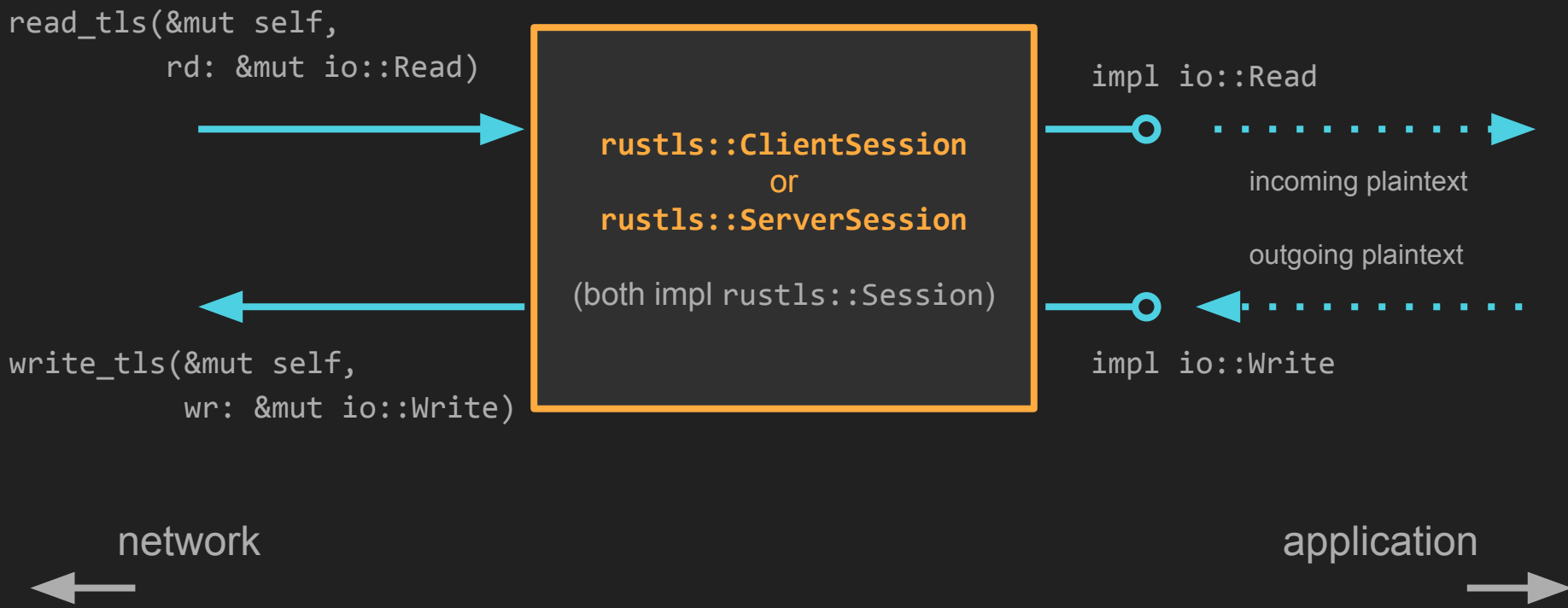
no obviously safe ciphersuites

- closest has design flaws that resulted in "Lucky13" vuln
- countermeasures possible, but unconvincing

long tail of old/misconfigured servers

early stages of death? (PCI from June 2018, Apple ATS, Cloudflare 'Require Modern TLS')

api


read_tls(&mut self,
rd: &mut io::Read)
rustls::ClientSession
or
rustls::ServerSession
(both impl rustls::Session)
impl io::Read
incoming plaintext
outgoing plaintext
impl io::Write
write_tls(&mut self,
wr: &mut io::Write)
network
application

api

```
pub trait Session: Read + Write + Send {

   fn read_tls(&mut self, rd: &mut Read) -> Result<usize, io::Error>;


   fn write_tls(&mut self, wr: &mut Write) -> Result<usize, io::Error>;


   fn process_new_packets(&mut self) -> Result<(), TLSError>;

(...)

}
```

api

to make a `rustls::ClientSession` you need a `rustls::ClientConfig` and the server's dns name

- one `ClientConfig` per process, typically
- contains root certificates
- plus a trait impl that persists data between sessions for resumption

similarly for `rustls::ServerSession` and `rustls::ServerConfig`

- `ServerConfig` contains a trait impl that chooses what certificate chain & private key to use for session

api usability

a `rustls::Session` buffers outgoing data if needed

- so you can send your HTTP request before the TLS handshake completes
- data only sent if handshake successful
- data sent in same flight as last message in handshake

the 'bring your own IO' interface has moderate usability cost

- allows compatibility with all IO models: non-blocking, thready, async

internals

msgs/*.rs: things that can de/serialise all the TLS types into convenient rust structs

```
Message { typ: Handshake, version: TLSv1_2, payload: Handshake(HandshakeMessagePayload
{ typ: ServerHello, payload: ServerHello(ServerHelloPayload { server_version: TLSv1_2,
random: Random([87, ..., 186]), session_id: SessionID, cipher_suite:
TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256, compression_method: Null, extensions:
[RenegotiationInfo(PayloadU8([])), ServerNameAcknowledgement,
SessionTicketAcknowledgement, ECPointFormats([Uncompressed])] }) }) }
```

internals

server_hs.rs/client_hs.rs: state machine for server/client

- complexity hotspot (~1300/~1500 lines each). ripe for refactoring.
- consists of functions of form:

```
fn handle_server_hello(sess: &mut ClientSessionImpl, m: Message) -> Result<&'static State, TLSError>
{ … }

pub static EXPECT_SERVER_HELLO: State = State {
    expect: Expectation {
        content_types: &[ContentType::Handshake],
        handshake_types: &[HandshakeType::ServerHello],
    },
    handle: handle_server_hello,
};
```

internals

- cipher.rs: TLS record layer encryption. a few implementations of these traits:

```
pub trait MessageDecrypter : Send + Sync {
    fn decrypt(&self, m: Message, seq: u64) -> Result<Message, TLSError>;
}

pub trait MessageEncrypter : Send + Sync {
    fn encrypt(&self, m: BorrowMessage, seq: u64) -> Result<Message, TLSError>;
}
```

testing

automated test suite made up of:

- integration tests against openssl and some public web servers
- api-level tests
- unit tests of library internals
- 'bogo' - the BoringSSL test suite, built from the golang TLS stack
  - hugely comprehensive and impressive
  - found some good bugs in rustls
- 'TryTLS' - python test suite concentrating on common implementation errors
- performance benchmarks

currently ~95% line coverage

performance

date	send speed (Gbps per core)	recv speed (Gbps per core)
2016-09-04	1.5	2.3
2016-09-20	2.0	6.5
2016-09-27	4.4	6.2
2017-01-26	15.9	15.0

all measurements taken on the same i5-6500 at 3.2GHz
suite: `TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256`

related crates

- webpki-roots: https://github.com/ctz/webpki-roots
  - It's the CA's in Mozilla's root program, suitable for ingestion by webpki
  - slightly harsher exclusions than implemented in NSS/Firefox
- mozilla-ca-certs: https://github.com/jld/mozilla-ca-certs
  - an alternative to webpki-roots, generated more straightforwardly than webpki-roots
  - slightly laxer exclusions than implemented in NSS/Firefox
- hyper-rustls: https://github.com/ctz/hyper-rustls
  - gluing hyper to rustls
  - would benefit from more use and testing

hyper-rustls

case study: github-rs

```
@@ -5,6 +5,8 @@ use hyper::header::{Accept, UserAgent, Authorization, Headers, qitem};
 use hyper::status::StatusCode;
+use hyper::net::HttpsConnector;
+use hyper_rustls::TlsClient;
 use types::*;

@@ -93,7 +95,7 @@ pub fn pagination(url: &mut String, page: Option<u64>, num_per:
Option<u64>) {

 /// GET requests
 pub fn get(url: &str, input_headers: Headers) -> Result<RawJSON> {
-    let client = Client::new();
+    let client = Client::with_connector(HttpsConnector::new(TlsClient::new()));
     let request = client.get(url);
     let mut buffer = String::new();
```

future work

- work out what to do with TLS1.3 0RTT data
- maybe support TLS1.1?
  - needs care, but not a great deal of work
- OCSP stapling support
- PSK support?
- RFC7627 extended master secret/session hash support
  - countermeasure for triple-handshake vulnerability
  - which rustls is not susceptible to, but it's just better

future work

- write some glue for use in tokio
- write some glue for use from non-rust programs
  - via libtls interface?
- find a better way to measure coverage
  - currently kcov, very very slow

thanks

Repo: https://github.com/ctz/rustls

Test server: https://rustls.jbp.io/

Twitter: @jpixton

Mail: jbp@jbp.io