FFI in Rust

Lessons from OpenSSL

Steven Fackler - sfackler

December 15, 2016

Example

```
struct foo {
    long a;
    char *b;
};
int fizzbuzz(const struct foo *f);
```

Example

```
struct foo {
    long a;
    char *b;
};
int fizzbuzz(const struct foo *f);
```

```
#[repr(C)]
struct foo {
    a: c_long,
    b: *mut c_char,
}
extern {
    fn fizzbuzz(f: *const foo) -> c_int;
}
```

openssl-sys

-sys?

A '-sys' crate provides a direct transcription of a C API into Rust - think of them as Rust's version of header files.

Also responsible for linking to the C library.

Bindgen

Bindgen uses libclang to parse C headers and produce Rust definitions from them.

```
$ bindgen fizzbuzz.h
/* automatically generated by rust-bindgen */

#![allow(dead_code,
         non_camel_case_types,
         non_upper_case_globals,
         non_snake_case)]
#[repr(C)]
#[derive(Copy, Clone)]
#[derive(Debug)]
pub struct foo {
    pub a: ::std::os::raw::c_long,
...
```

Caveats

While Bindgen is fantastic to quickly get Rust bindings for a header, it has some drawbacks:

- libclang is a fairly heavyweight dependency.
- It can be a bit verbose, particularly when system headers are included.
- It produces definitions valid in the environment it's run.

rust-openssl does not use bindgen for its bindings.

The Manual Approach

You will get things wrong if you try to do it by hand.

```
extern {
    pub fn SSL_CTX_use_certificate_file(
        ctx: *mut SSL_CTX,
        cert_file: *const c_char,
        file_type: c_int)
        -> c_int;

    pub fn SSL_CTX_use_certificate_chain_file(
        ctx: *mut SSL_CTX,
        cert_chain_file: *const c_char,
        file_type: c_int)
        -> c_int;
}
```

The Manual Approach

You will get things wrong if you try to do it by hand.

```
extern {
    pub fn SSL_CTX_use_certificate_file(
        ctx: *mut SSL_CTX,
        cert_file: *const c_char,
        file_type: c_int)
        -> c_int;

    pub fn SSL_CTX_use_certificate_chain_file(
        ctx: *mut SSL_CTX,
        cert_chain_file: *const c_char,
        file_type: c_int) // OOPS!
        -> c_int;
}
```

ctest

The ‘ctest’ crate will check that your bindings correctly correspond to the C definitions.

```
error: incompatible pointer types returning 'int
    (SSL_CTX *, const char *)' (aka 'int (struct
    ssl_ctx_st *, const char *)') from a function
    with result type 'int (*)(SSL_CTX *, const
    char *, int)' (aka 'int (*)(struct ssl_ctx_st *,
    const char *, int)')
    [-Werror,-Wincompatible-pointer-types]

    return SSL_CTX_use_certificate_chain_file;
           ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

Version/Feature Detection

OpenSSL releases are not binary compatible, but 1.0.1, 1.0.2, and 1.1.0 are all supported by rust-openssl. Features have been added and removed; structs have been made opaque.

OpenSSL's feature set can be customized at compile time. Constants, functions, and even struct fields are removed!

```
struct ssl_ctx_st {
    // ...
    int references;
    // ...
#   ifndef OPENSSL_NO_TLSEXT
    unsigned alpn_client_proto_list_len;
    // ...
#   endif
    // ...
};
```

Version/Feature Detection

The `opensslv.h` header contains `#defines` for the version and all compile time feature flags. Parse it in a build script and turn those into Rust cfgs!

```
#[repr(C)]
pub struct SSL_CTX {
    // ...
    pub references: c_int,
    // ...
    #[cfg(all(not(osslconf = "OPENSSL_NO_TLSEXT"),
              ossl102))]
    alpn_client_proto_list_len: c_uint,
    // ...
}
```

openssl

Ownership

Figuring it Out

Ownership semantics are as important to C libraries as they are in Rust, but handled implicitly.

If you're lucky, your library defines ownership conventions:

- OpenSSL - get1/get0, set1/set0, add1/add0
- Core Foundation - The Create Rule

Figuring it Out

These are probably not enough.

The majority of OpenSSL's APIs do not use the get1/get0 convention!

Some functions are special snowflakes - `X509_STORE_add_cert` takes ownership of its argument even when an error is encountered.

Figuring it Out

These are probably not enough.

The majority of OpenSSL's APIs do not use the get1/get0 convention!

Some functions are special snowflakes - `X509_STORE_add_cert` takes ownership of its argument even when an error is encountered.

The Source is the Only Source of Truth.

Translating

openssl takes a dual-type approach - think `Path`/`PathBuf`:

```
pub struct X509Ref(UnsafeCell<()>);

pub struct X509(*mut ffi::X509);

impl Drop for X509 { /* ... */ }

impl Deref for X509 {
    type Target = X509Ref;
    // ...
}

impl DerefMut for X509 { /* ... */ }
```

Fun With Pointers

```
impl X509Ref {
    pub unsafe fn from_ptr<'a>(ptr: *mut ffi::X509)
                                    -> &'a X509Ref {
        &*(ptr as *mut _)
    }

    pub fn as_ptr(&self) -> *mut ffi::X509 {
        self as *const _ as *mut _
    }
}

impl Deref for X509 {
    fn deref(&self) -> &X509Ref {
        unsafe { X509Ref::from_ptr(self.0) }
    }
}
```

Enums

Rust enums are not like C enums!

Adding to a Rust enum is not backwards compatible.

What do you do when C returns a value you don't expect?

```
pub struct Padding(c_int);

pub const PKCS1_PADDING: Padding =
    Padding(ffi::RSA_PKCS1_PADDING);
```

Panic Tunnelling

Never forget about panics! Unwinding through non-Rust is undefined behavior.

```
int foobar(int (*cb)(void *), void *data);
```

```
fn foobar<F>(f: F) -> Result<(), Error>
    where F: FnMut();
```

Panic Tunnelling

```
struct Callback<F> { f: F, p: Option<Box<Any+Send+'static>> }

unsafe extern fn raw_cb<F>(data: *mut c_void) -> c_int
    where F: FnMut()
{
    let data: &mut Callback<F> = &*(data as *mut _);

    match panic::catch_unwind(AssertUnwindSafe(|| (data.f)()) {
        Ok(()) => 1,
        Err(err) => {
            data.panic = Some(err);
            0
        }
    }
}
```

Panic Tunnelling

```
fn foobar<F>(f: F) Result<(), Error>
    where F: FnMut()
{
    let mut cb = Callback { f: f, p: None };
    let r = ffi::foobar(raw_cb::<F>,
                        &mut cb as *mut _ as *mut _);
    if r == 1 {
        Ok(())
    } else {
        if let Some(err) = callback.p {
            panic::resume_unwind(p);
        } else {
            Err(Error(...))
        }
    }
}
```

Version/Feature Detection

Recall:

- Three versions of OpenSSL supported. Each adds and removes functionality.
- Features can be disabled at OpenSSL build time.

Approach:

- Cargo features for each OpenSSL version. Each enables functionality exposed in that version *when linking against it*. Crates can ask openssl-sys which version has been detected in a build script.
- Features that are disabled by OpenSSL build flags are simply removed when those flags are set.

Stay True to the Library

It should be obvious to people familiar with the C library how to use your layer.

Lean towards 1:1 mappings from things in the C library to your crate.

There's a somewhat fine line to walk between Rustifying the API and burying it too deep.

... But Not Always

OpenSSL's TLS configuration is a minefield.

- Default cipher list is overly permissive (RC4!).
- SSLv2 and SSLv3 are enabled by default (maybe).
- Ephemeral key exchange is disabled by default. Each supported version requires different steps to enable ECDHE!
- Certificate validation is disabled entirely by default.
- Hostname validation is enabled separately and doesn't even exist in 1.0.1.

... But Not Always

If you are using an `SslContext` directly, you are probably doing it wrong.

... But Not Always

If you are using an `SslContext` directly, you are probably doing it wrong.

Clients - `SslConnector` provides a configuration modeled after Python 3.6's.

Servers - `SslAcceptor` provides configurations modeled after the modern and intermediate profiles of Mozilla's Server Side TLS recommendations.

Wrapup

Questions?

sfackler in #rust

`https://github.com/sfackler/rust-openssl`

`https://github.com/alexcrichton/ctest`

`https://wiki.mozilla.org/Security/Server_Side_TLS`