Firefox

Oblivious HTTP in Firefox

What we don't know can't hurt you

30.03.23

Dana Keeler — Senior Staff Security Engineer

Table of contents

Hi

01

Biographical Details

Dana Keeler (she/her)

Tech lead for Oblivious HTTP in Firefox

Security engineer at Mozilla

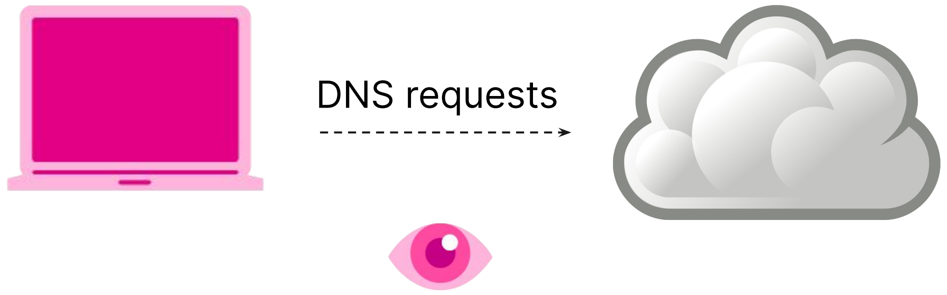
@mozkeeler on Twitter

Oblivious HTTP

02

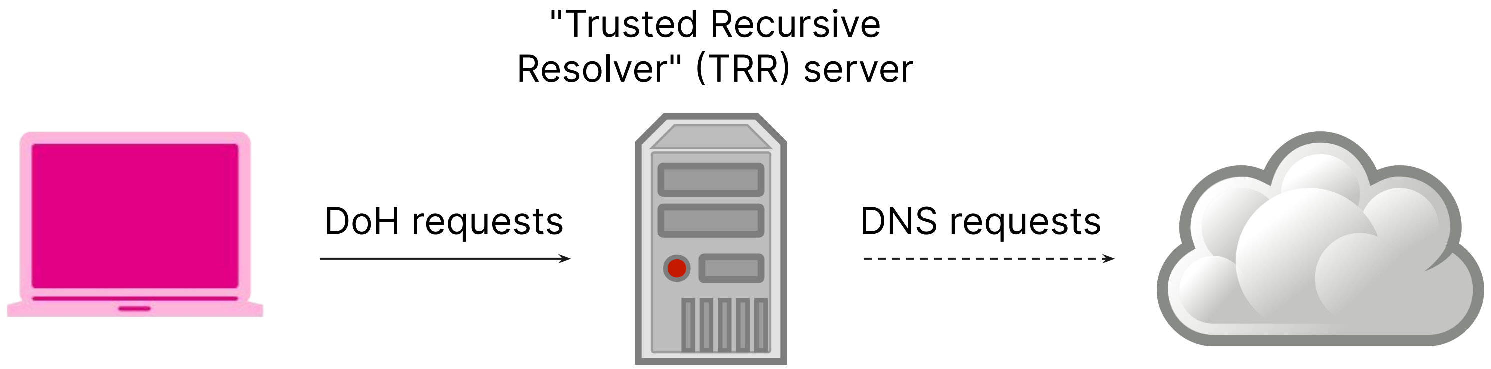
Motivation

DNS



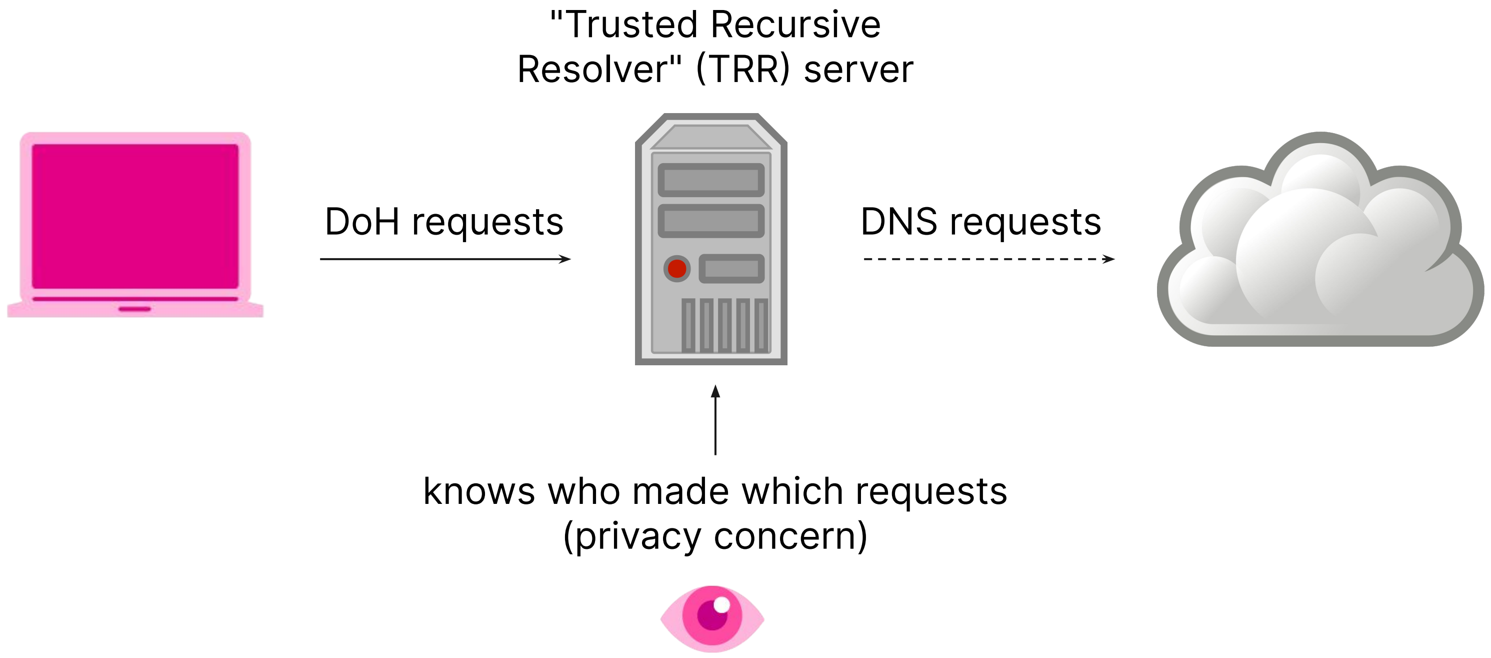
Motivation

DNS-over-HTTPS (DoH)



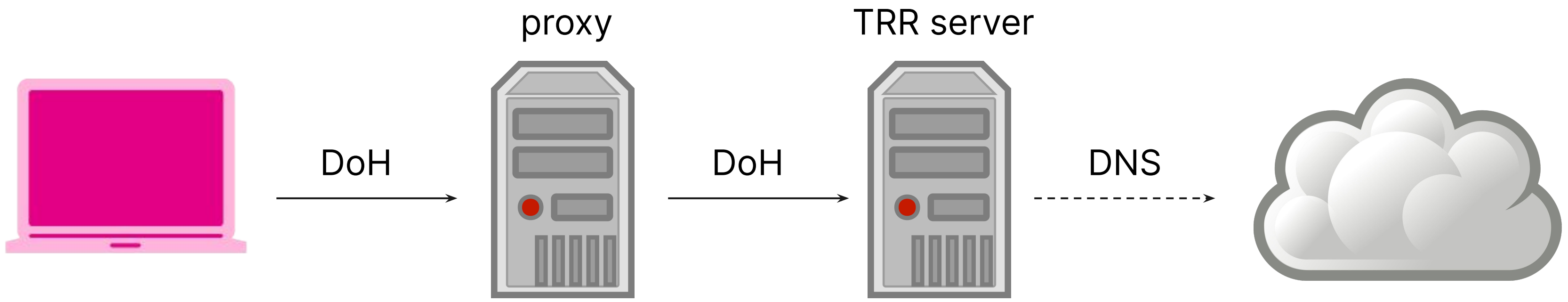
Motivation

DNS-over-HTTPS (DoH)


"Trusted Recursive Resolver" (TRR) server
DoH requests
DNS requests
knows who made which requests (privacy concern)

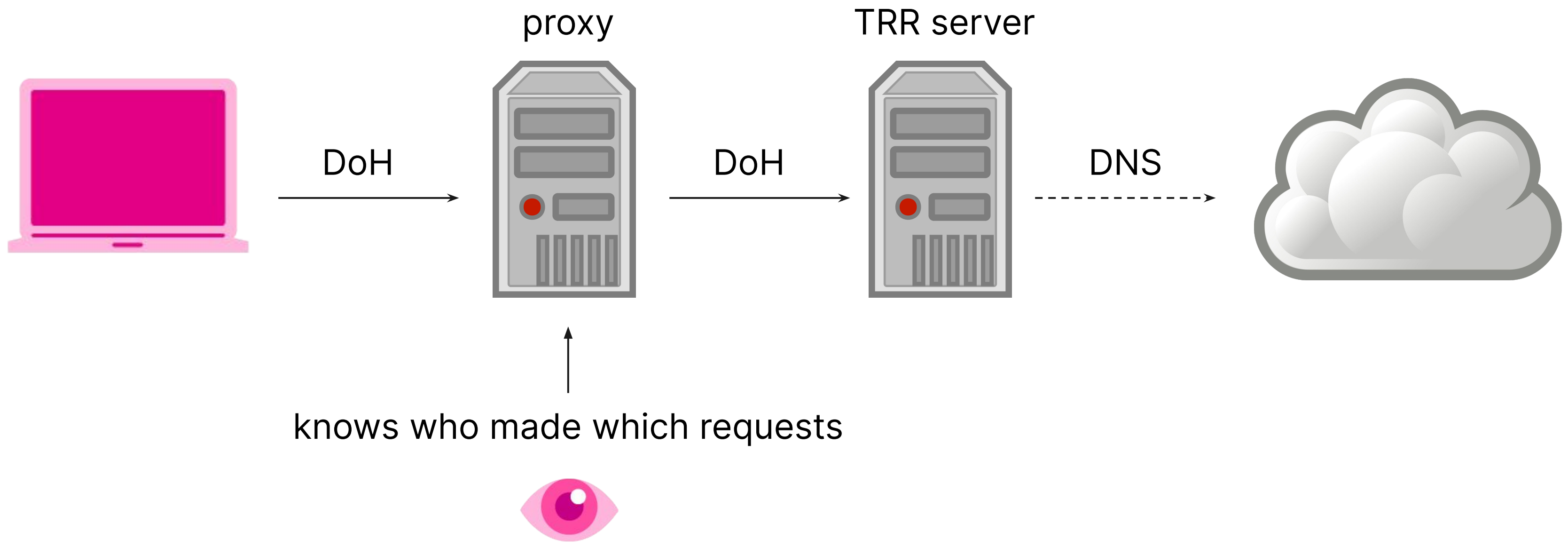
Motivation

DNS-over-HTTPS: With a proxy?



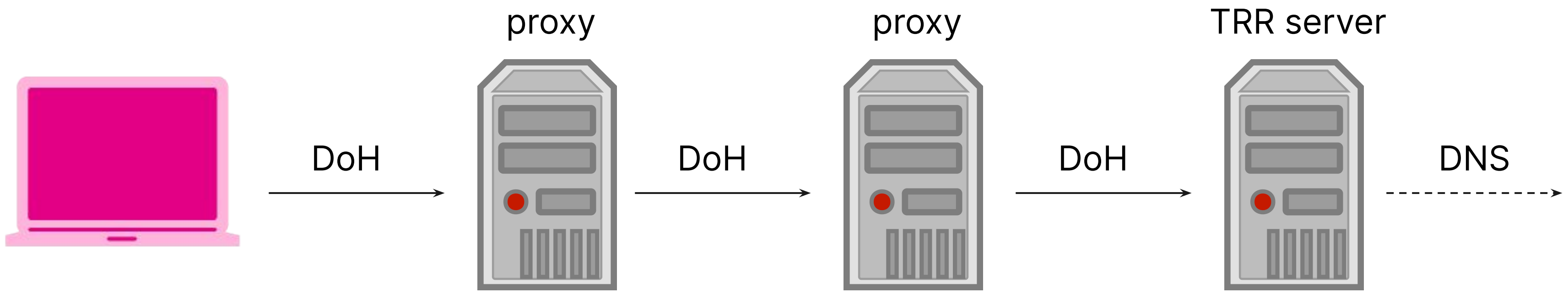
Motivation

DNS-over-HTTPS: With a proxy?



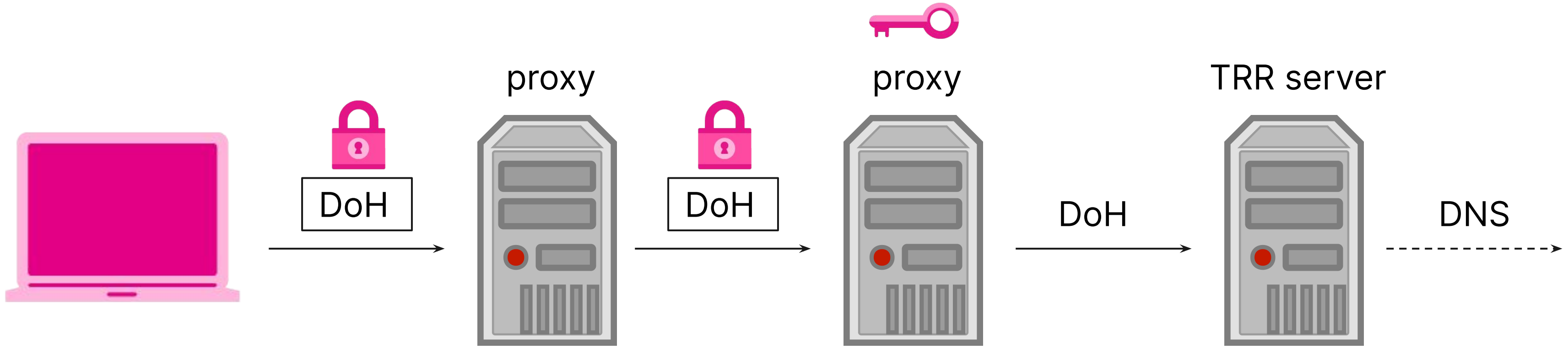
Motivation

DNS-over-HTTPS: Two-hop proxy?



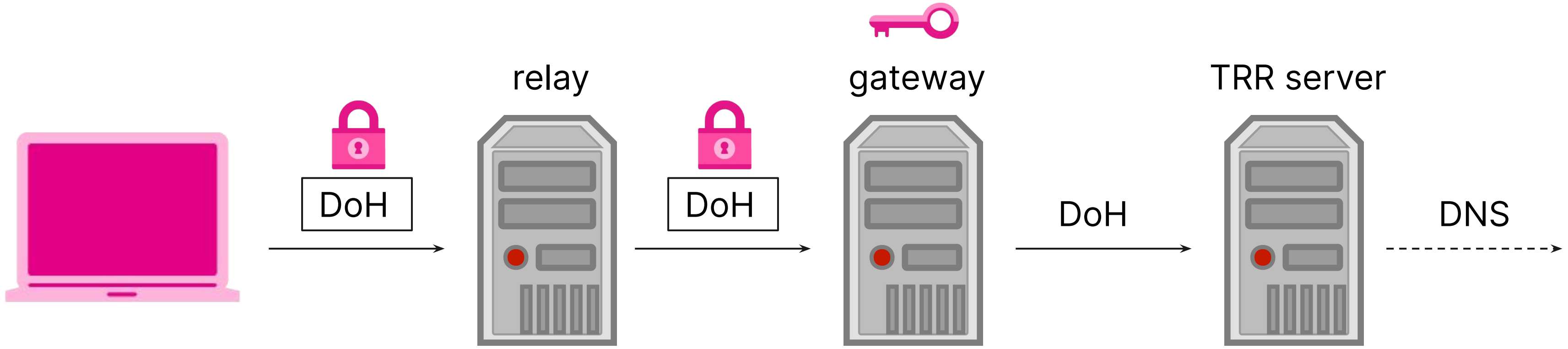
Motivation

DNS-over-HTTPS: Two-hop proxy?



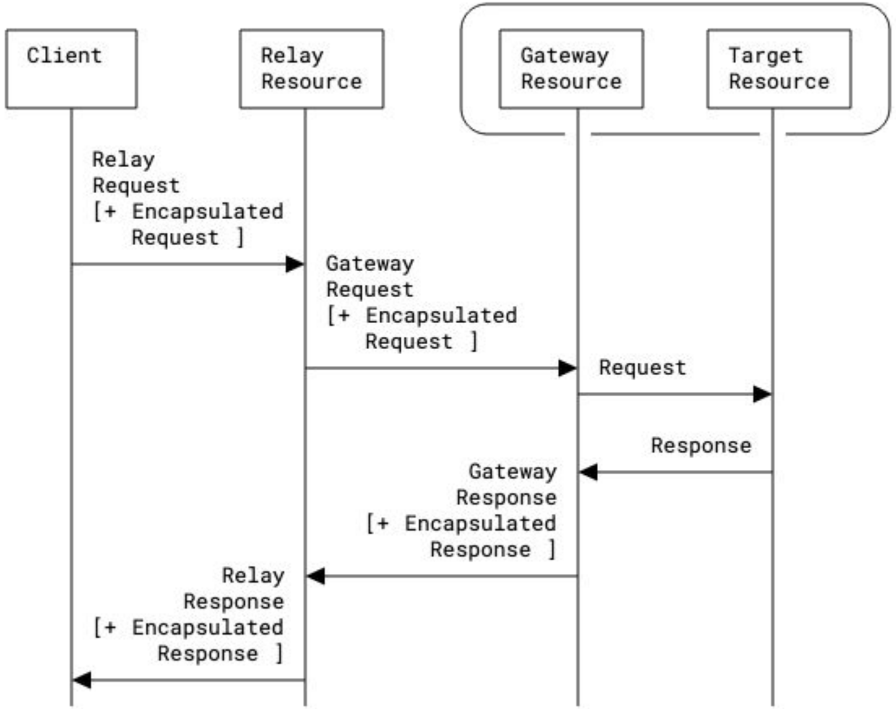
Motivation

DNS-over-HTTPS: Two-hop proxy?



Oblivious HTTP

Hybrid Public Key Encryption (HPKE - RFC 9180) + 2-hop proxy = OHTTP


Client
Relay Resource
Gateway Resource
Target Resource
Relay Request [+ Encapsulated Request]
Gateway Request [+ Encapsulated Request]
Request
Response
Gateway Response [+ Encapsulated Response]
Relay Response [+ Encapsulated Response]

https://datatracker.ietf.org/doc/draft-ietf-ohai-ohttp/

Implementation Considerations

03

Language

What language do we use?

C++

+ Much of Firefox is C++
+ In particular, Necko (the networking engine) is C++
– What about cryptography?

C

+ Firefox's cryptography library, NSS, is C
– Will not help you avoid mistakes (memory safety, type safety, thread safety, etc.)

Rust

+ Powerful and expressive; will help you avoid mistakes (addresses memory, type, and thread safety)
– Lack of well-vetted cryptography implementations that aren't just OpenSSL

Language

What language do we use?

Why not all three?

C++ for incorporating into Necko

Rust for implementing the logic of Oblivious HTTP

C for cryptography

Multi-Language Drifting

How even

Calling C from Rust

Rust has Foreign Function Interface (FFI) support out-of-the-box:

```
    extern "C" { fn some_c_function(buf: *mut c_char, len: c_ulong); }
```

… but hand-crafting FFI declarations is tedious and error-prone

So we use bindgen!

Bindgen programmatically generates Rust declarations given C header files

Multi-Language Drifting

How even

Calling C from Rust: Generating Bindings

```
// build.rs
fn main() {
  let bindings = Builder::default()
    .header("wrapper.h")
    .allowlist_function("PK11_HPKE_Seal")
    .allowlist_type("SECItem")
    .generate().expect("Unable to generate bindings");
  let out_path = PathBuf::from(env::var("OUT_DIR").expect("OUT_DIR not set?"));
  bindings.write_to_file(out_path.join("bindings.rs")).expect("Couldn't write");
}
```

Multi-Language Drifting

How even

Calling C from Rust: Generating Bindings

```
// wrapper.h
#include "secoidt.h"
#include "keyhi.h"
#include "pk11pub.h"
```

Multi-Language Drifting

How even

Calling C from Rust: Generating Bindings

```
// bindings.rs
#[repr(C)]
pub struct SECItem {
     pub type_: SECItemType::Type,
     pub data: *mut ::std::os::raw::c_uchar,
     pub len: ::std::os::raw::c_uint,
}
extern "C" {
  pub fn PK11_HPKE_Seal(cx: *mut HpkeContext, aad: *const SECItem,
    pt: *const SECItem, outCt: *mut *mut SECItem) -> SECStatus;
}
```

Multi-Language Drifting

How even

Calling C from Rust

```
// https://github.com/martinthomson/ohttp/blob/main/ohttp/src/nss/hpke.rs
pub fn seal(&mut self, aad: &[u8], pt: &[u8]) -> Res<Vec<u8>> {
  let mut out: *mut sys::SECItem = null_mut();
  secstatus_to_res(unsafe {
    sys::PK11_HPKE_Seal(*self.context, &Item::wrap(aad), &Item::wrap(pt), &mut out)
  })?;
  let v = Item::from_ptr(out)?;
  Ok(unsafe { v.into_vec() })
}
```

Multi-Language Drifting

How even

Calling Rust from C++

Again, manual C FFI declarations are tedious

Could use bindgen...

Firefox has a better fit: XPCOM (Mozilla's Cross-Platform Component Object Model)!

Historically for calling C++ from JavaScript and vice-versa

As of a few years ago, Firefox supports calling Rust to and from C++ and JavaScript

Calling Rust from C++: Defining an Interface

```
interface nsIObliviousHttp : nsISupports {
  nsIObliviousHttpClientRequest encapsulateRequest( // encrypt a request
    in Array<octet> encodedConfig, // HPKE configuration of gateway
    in Array<octet> request); // encoded request
};

interface nsIObliviousHttpClientRequest : nsISupports {
  readonly attribute Array<octet> encRequest; // the encrypted request
  readonly attribute nsIObliviousHttpClientResponse response; // context to decrypt response
};

interface nsIObliviousHttpClientResponse : nsISupports {
  Array<octet> decapsulate(in Array<octet> encResponse); // decrypt an encrypted response
};
```

Calling Rust from C++: Defining an Interface

```
#[repr(C)]
pub struct nsIObliviousHttp {
  vtable: *const nsIObliviousHttpVTable
}

impl nsIObliviousHttp {
  pub unsafe fn EncapsulateRequest
    &self,
    encodedConfig: *const ThinVec<u8>,
    request: *const ThinVec<u8>,
    _retval: *mut *const nsIObliviousHttpClientRequest,
  ) -> nsresult {
    ((*self.vtable).EncapsulateRequest)(self, encodedConfig, request, _retval)
  }
}
```

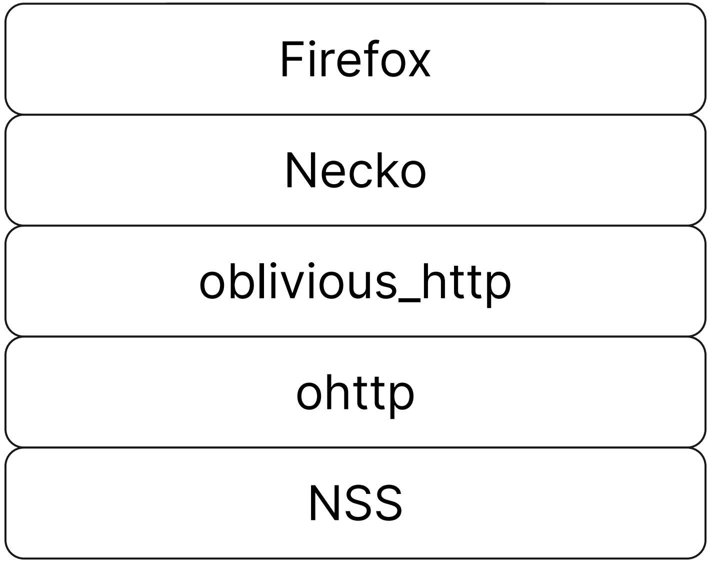
Calling Rust from C++: Implementing the Interface

```
extern crate ohttp;
use ohttp::{ ClientRequest, ClientResponse, KeyConfig, ... };

#[xpcom(implement(nsIObliviousHttp), atomic)]
struct ObliviousHttp {}

impl ObliviousHttp {
  xpcom_method!(encapsulate_request => EncapsulateRequest(encoded_config: *const ThinVec<u8>,
    request: *const ThinVec<u8>) -> *const nsIObliviousHttpClientRequest);
  fn encapsulate_request(&self, encoded_config: &ThinVec<u8>, request: &ThinVec<u8>
  ) -> Result<RefPtr<nsIObliviousHttpClientRequest>, nsresult> {
    let client = ClientRequest::new(encoded_config).map_err(|_| NS_ERROR_FAILURE)?;
    let (enc_request, response) = client.encapsulate(request).map_err(|_| NS_ERROR_FAILURE)?;
    ...
```

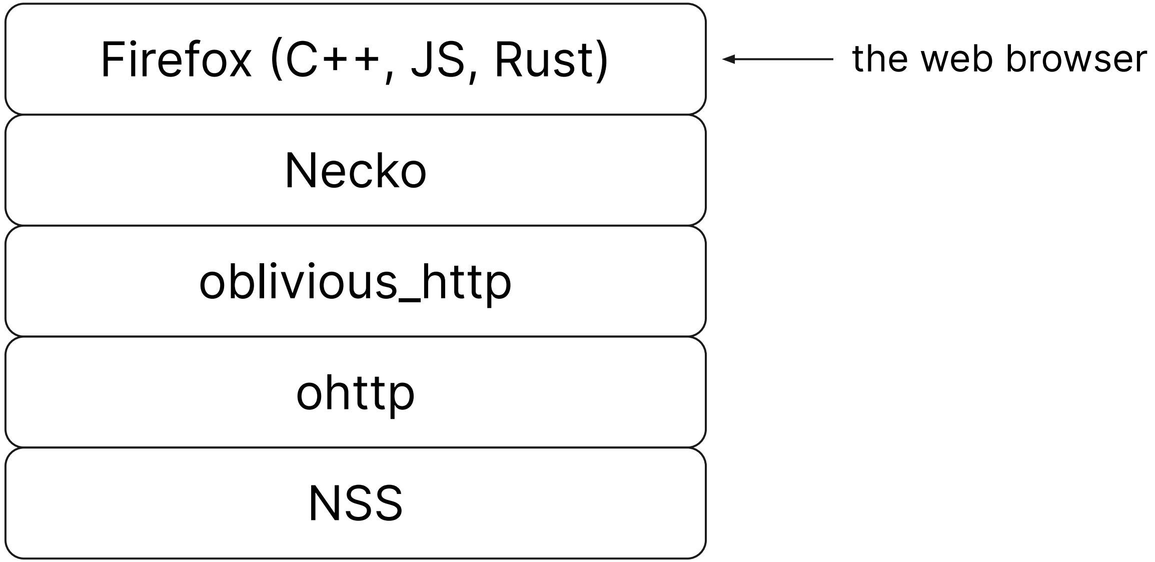
Calling Rust from C++: Implementing the Interface

```
#[xpcom(implement(nsIObliviousHttpClientRequest), atomic)]
struct ObliviousHttpClientRequest {
  enc_request: Vec<u8>,
  response: RefPtr<nsIObliviousHttpClientResponse>,
}

impl ObliviousHttpClientRequest {
  xpcom_method!(get_enc_request => GetEncRequest() -> ThinVec<u8>);
  fn get_enc_request(&self) -> Result<ThinVec<u8>, nsresult> {
    Ok(self.enc_request.clone().into_iter().collect())
  }

  xpcom_method!(get_response => GetResponse() -> *const nsIObliviousHttpClientResponse);
  fn get_response(&self) -> Result<RefPtr<nsIObliviousHttpClientResponse>, nsresult> {
    Ok(self.response.clone())
  }
}
```

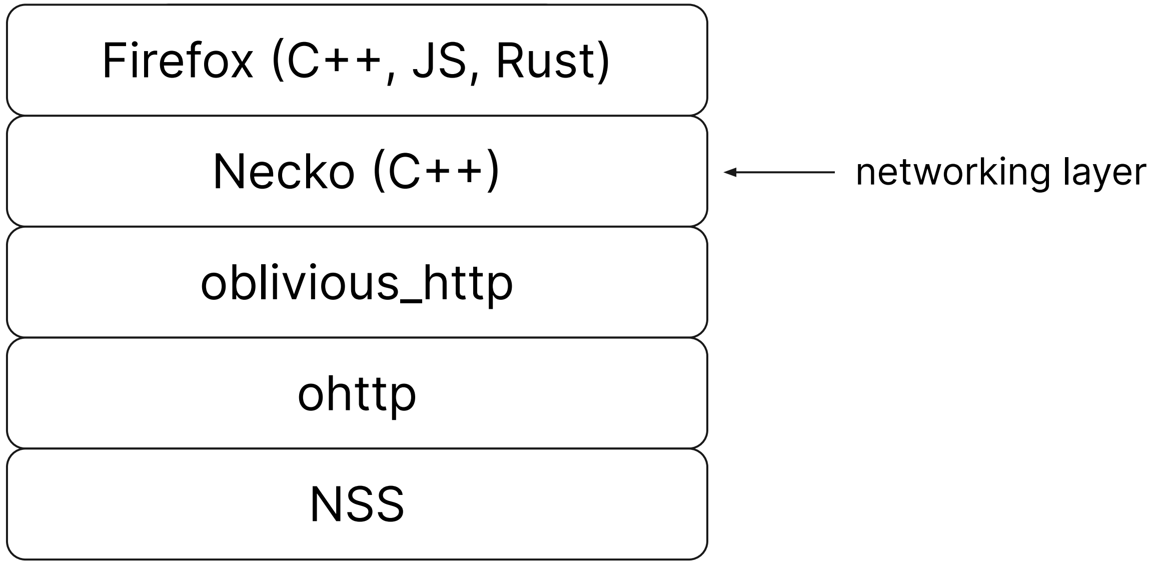
All Together Now



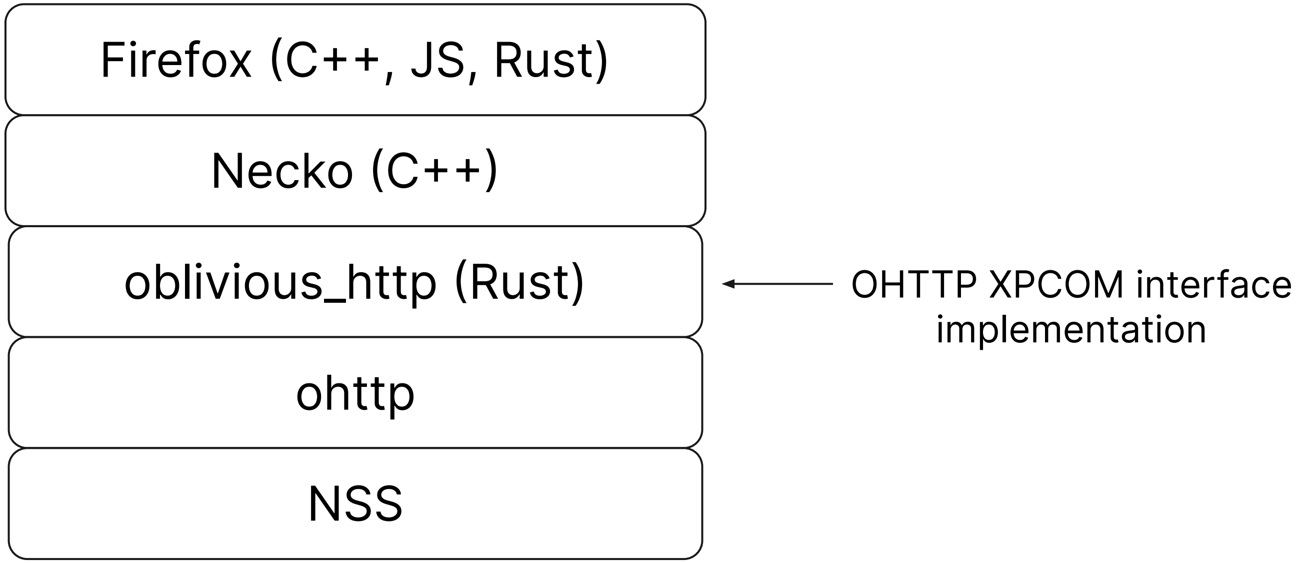
All Together Now



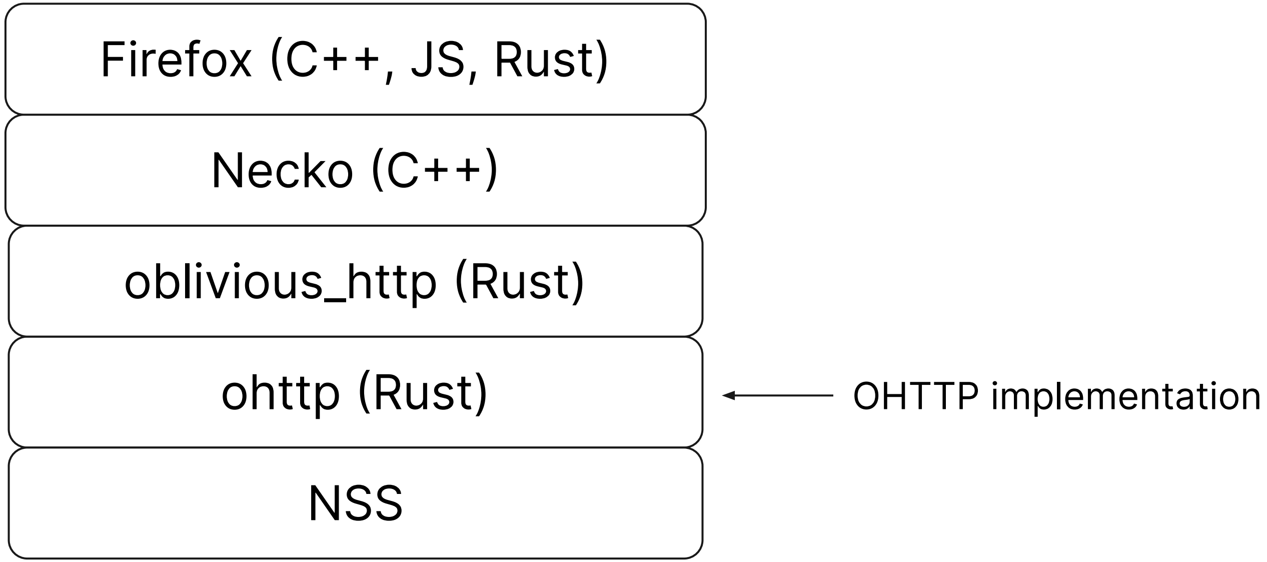
All Together Now



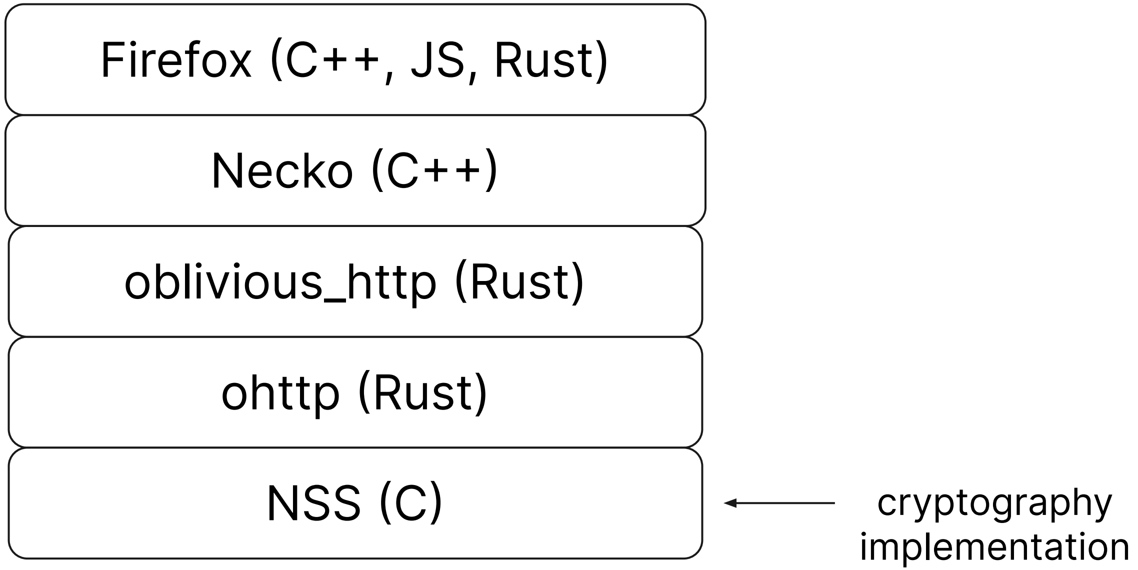
All Together Now



All Together Now



All Together Now



Thank you!