TLS Encrypted Client Hello (ECH)

This Document is Public

Authors: dmcardle@chromium.org

10/2020

> *This is a living document. The current focus is ECH in **BoringSSL**.*
>
> *Over time, we will flesh out the details on*
> - *Chromium integration*
> - *Chromium experiment and rollout*
> - *QUIC integration.*

One-page overview

Summary

The TLS Encrypted ClientHello (ECH) extension enables clients to encrypt ClientHello messages, which are normally sent in cleartext, under a server's public key. This avoids leaking sensitive fields like the server name to the network. ECH is currently specified in draft-ietf-tls-esni-13. Note that earlier iterations of this specification were called Encrypted Server Name Indication, or ESNI.

Platforms

All Blink platforms.

Team

dmcardle@chromium.org, davidben@chromium.org, kenjibaheux@chromium.org

Bug

Chromium: https://crbug.com/1091403
BoringSSL: https://crbug.com/boringssl/275

Code affected

Network stack, BoringSSL

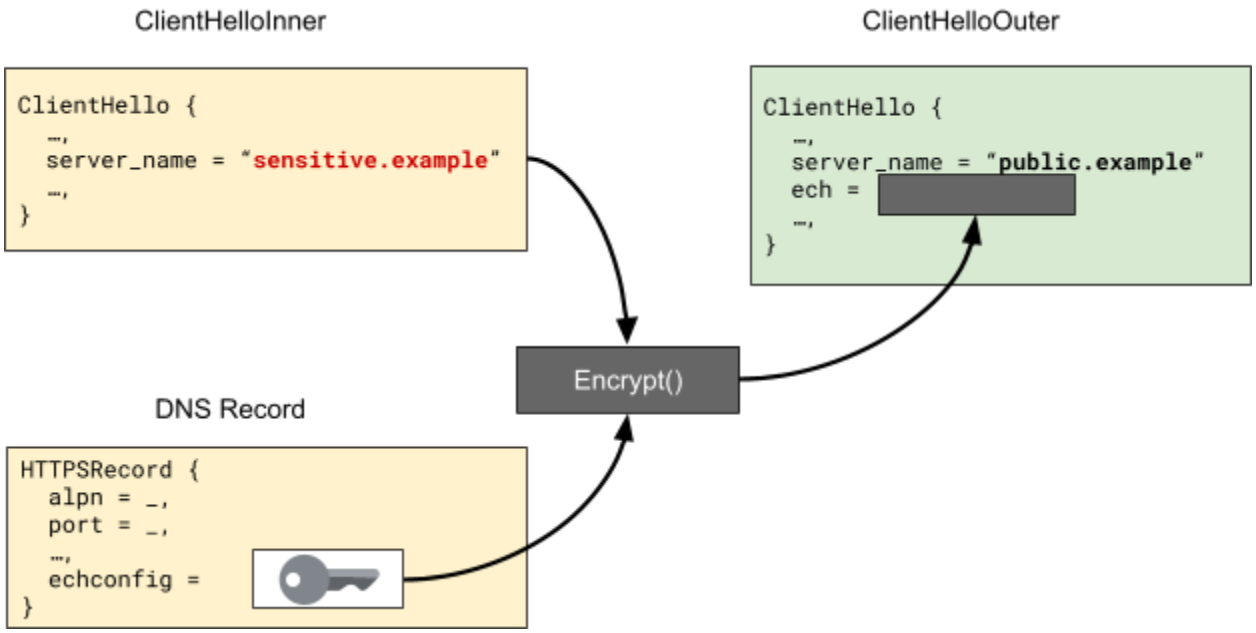
Design

Background

Today, when a client initiates a TLS connection with a server, it sends an unencrypted ClientHello message containing the server name and other potentially sensitive fields. This reveals information about the user's browsing to network adversaries.

TLS Encrypted Client Hello (ECH) enables clients to encrypts the ClientHello message using an `ECHConfigList` obtained from DNS[1] HTTPS records. The `ECHConfigList` contains the server's public keys and other metadata. See also DNS HTTPS Records.

Given an `ECHConfigList`, the client encrypts the true "ClientHelloInner" and inserts it into an `encrypted_client_hello` extension the "ClientHelloOuter". The ClientHelloOuter's (cleartext) fields should not contain sensitive information.



Constructing the ClientHelloOuter.

[1] Note these keys are only as trusted as the connection to the DNS provider. If using DNS over HTTPS, the keys are as trusted as the DNS provider. If using cleartext DNS, network attackers on path to the DNS provider can also inject keys. Note, however, the server name is also sent to the DNS provider.

The server then decrypts the ClientHelloInner with its corresponding private keys and completes the handshake as if ClientHelloInner was sent. On key mismatch or rollback, the server completes the handshake with ClientHelloOuter, which acts as an authenticated recovery mechanism.

When Chrome cannot obtain an `ECHConfigList` for a server, it will send a normal ClientHello message with a GREASE ECH extension.

BoringSSL

The initial experiment target is ECH draft-13. On the specification side, there are still some open questions around padding the server response, particular with relation to QUIC. Initially, we'll skip server padding, but the final implementation and protocol will incorporate it in some form.

ECH depends on the Hybrid Public Key Encryption (HPKE) primitive. This has been implemented in BoringSSL. ECH draft-13 has also been implemented in BoringSSL. The API is documented here.

Design details:

- TLS's key schedule uses a handshake transcript (see RFC 8446 § 7.1), implemented by BoringSSL's SSLTranscript class. On the client, it's not initially known whether ClientHelloOuter or ClientHelloInner will be used. We will maintain a second parallel transcript using the ClientHelloInner. The existing SSL_HANDSHAKE::transcript will use the ClientHelloOuter, while a new inner_transcript will use the ClientHelloInner. Once the ClientHello is known, we will replace SSLTranscript::transcript with inner_transcript as needed.

- The server does not maintain two handshakes, but it does use a different ClientHello. When the server accepts ECH, we retain the ClientHelloInner on SSL_HANDSHAKE and read from it instead of the ClientHelloOuter.

- BoringSSL enables servers to set a callback with SSL_CTX_set_select_certificate_cb, called after receiving the ClientHello. We will process ECH before this callback is invoked, so application code selects the certificate, etc., appropriately. Likewise, SSL_get_servername will return information from the selected ClientHello.

- To avoid duplicating large extensions that are already present on the outer ClientHello, ECH enables ClientHelloInner to point to the ClientHelloOuter's extensions with the "outer_extensions" extension. We've made the add_clienthello

callbacks const, so they can be called twice, once for each ClientHello. The callbacks additionally specify whether the extension is “compressible”, which indicates they match between the two ClientHellos and it is public that they do. If so, it is safe to compress the extension body with “outer_extensions”.

- We can omit TLS-1.2-only extensions from the ClientHelloInner.

- On the client, if handshaking with ClientHelloOuter, we must authenticate with the public name. The caller needs an API to modify its certificate verification. We must also disable client certificates, as those should only be sent to the true name. Finally, we must fail the handshake on completion, and instead provide an API to pass retry configs to the caller.

Chrome

Most of ECH in Chrome will be plumbing between the DNS logic and BoringSSL. The `EndpointServiceConnectionInfo` structure, to be added as part of HTTPS record work, will contain the serialized `ECHConfigList`. `SSLConnectJob` will read this value and pass it into `SSLClientSocketImpl` via a corresponding field in `SSLConfig`. Note it is important to use an `ECHConfigList` that matches the route chosen. Depending on how DNS and `TransportConnectJob` interact, we may need to plumb some information out.

From there, `SSLClientSocketImpl` will configure ECH in BoringSSL if the field is present in `ECHConfigList`. It will also enable ECH GREASE, in case no `ECHConfig` was suitable, or the field was missing. If ECH is accepted, the connection then proceeds as before.

To handle ECH rejection, `SSLClientSocketImpl` will call `SSL_get0_ech_name_override` to verify with the public name if needed. When verifying with the public name, we will map certificate errors to a new unbypassable error. Like certificate errors from proxies and DoH, we will not make ClientHelloOuter certificate errors bypassable. This avoids prompting users about a bad certificate for a name other than the one they are connecting to. However, we should trigger the captive portal detector and bad clock logic, if possible.

It will also map `SSL_R_ECH_REJECTED` to a corresponding error code in //net, and report the retry configs out of a `GetEchRetryConfigs` access.

The first time `SSLConnectJob` sees an ECH reject, it will call `GetEchRetryConfigs` and retry the connection with the new value. This implements the recovery flow on key mismatches.

We will not implement any separate ECHConfig cache and instead rely on DNS caching. (An external ECHConfig cache will not interact correctly with multi-CDN use cases.)

TODO(davidben): Map out how to implement this on the QUIC side as well.

TODO(davidben): Enterprise policy plans.

🚧 *This rest of this document is under construction.* 🚧

Metrics

Success metrics

You should list what metrics you will be tracking to measure the success of your feature or change. This could be a mix of existing and new metrics. If they are new metrics, explain how they will work. If you aim to improve performance with your feature or change, you should measure your impact on one of the speed launch metrics.

Regression metrics

You should define what metrics you will be tracking to look for potential regressions associated with your feature or change. This could be a mix of existing and new metrics. The speed launch metrics are good candidates for use as performance regression metrics. If you're using new metrics, explain how they will work.

Experiments

If you are using Finch to run experiments (see go/newChromeFeature for advice), describe what experiments you intend to run and what you are looking for in the results. It's important that you know in advance what the acceptance criteria are. List the experiment names so people can look them up (links to the dashboard are even better).

The ECH experiment's Finch configuration will define a set of domains for which Chrome will attempt ECH. If the HTTPS DNS experiment is in place, this ECH experiment can obtain the server's `ECHConfigs` through DNS. Otherwise, we can fall back on a "*.well-known*" location for key delivery (only for experimentation, as this negates the point of encrypting SNI).

Rollout plan

If you're just checking in the code and doing a dev-beta-stable progression, just write "Waterfall" here. If you're doing a standard experiment-controlled rollout, write that with the experiment name. If you're not doing the standard rollout, describe what you're doing and why it needs to be different.

If there are external deadlines, call them out (if you don't want these to be public, use the internal template). Otherwise, releases should be quality driven and you shouldn't be targeting a milestone.

Core principle considerations

Everything we do should be aligned with and consider Chrome's core principles. If there are any specific stability concerns, be sure to address them with appropriate experiments.

Speed

Describe the considerations you're making with respect to how this work impacts Chrome's performance (speed, memory usage, power, etc.). Note that no change should regress the speed launch metrics.

It can be very hard to predict the end-user impact of a change on performance due to the wide variety of web content, device types, network connections and other factors in the field. Therefore, speed releasing team strongly recommends that finch is used for any launch that could plausibly affect speed. Early indicators of performance can be seen by running benchmarks on the perf trybots or cluster telemetry, but ideally performance impact would be measured by the speed launch metrics on end users.

Security

Sketch your threat model and describe the system's security mechanisms, especially around the handling of untrusted data. Be sure to describe any attack surfaces, any known vulnerabilities or points of failures, as well as any potentially insecure dependencies. If your feature doesn't have security considerations, explicitly state so (and why).

Privacy considerations

Features with privacy implications should use the internal template for privacy review.

Authenticity of DNS HTTPS Records

Chrome has no proof that `HTTPS` records and the `ECHConfigs` they contain are authentic. (Although records should be authenticated at the DoH resolver.)

A malicious DoH server could disable ECH for a client by dropping `ECHConfigs` from the response. Given that the point of ECH is to protect the server name, which DNS already knows, the resolver does not have much to gain from this attack.

Alternatively, a malicious DoH server could serve uniquely-identifying `ECHConfigs` to each client. When the client uses the spurious `ECHConfigs` to establish a TLS connection to the server, it would send a uniquely-identifying `ECHConfigs.record_digest` value to the server. In this way, a DNS-level identifier could leak into third-party contexts such as an iframe. However, a similarly powerful attack could be performed by tampering with the A/AAAA responses. This attack is not unique to ECH, so we will consider it out of scope.

Testing plan

It goes without saying that all code should have good tests run on the waterfall. You don't need to write about that.

Here is where you should describe what the test team may need to consider before approving your launch. Some features won't need special testing considerations. If so, say this and why. Otherwise, give any directions needed to exercise your feature. Call out any special platforms conditions that may require extra attention.

BoringSSL

BoringSSL tests its TLS implementation with a test suite written in Go. We have implemented ECH in the Go implementation and written various tests.

One detail of note is, in the Go implementation, the extensions to be compressed are configurable. This allows us to test the C implementation with a variety of inputs. The C implementation does not need such a generic capability.

Test Cases for "outer_extensions" Extension

- Test the server rejects ClientHelloOuter with "outer_extensions" extension.
- Test the server rejects ClientHelloInner with malformed "outer_extensions" extension.
- Test the server accepts ClientHelloInner with valid "outer_extensions" extension (referencing zero and non-zero number of outer extensions that really exist in ClientHelloOuter).
- Test the server rejects when ClientHelloInner references nonexistent outer extension.
- Test the server rejects when ClientHelloInner references outer extension that already exists in inner extensions.

Fuzzing

ECH has been integrated into BoringSSL's "fuzzer mode". When fuzzing, we skip the encryption portion of ECH, so the fuzzer can discover interesting ECH-related cases. We also have a standalone fuzzer for the ClientHelloInner decoder, to cover the outer_extensions logic.

Chromium

Followup work

How will you assess the success of this work? What needs to be followed-up on? What (experimental code, for example) needs to be cleaned up after the code has reached the stable channel?