DNS HTTPS Records

This Document is Public

Author: ericorth@chromium.org
2021-04

One-page overview

Summary

Query and parse DNS HTTPS records (previously known as HTTPSSVC records) alongside the traditional A and AAAA records. Use information from these records to improve privacy and performance of HTTPS web connections.

Platforms

Mac, Windows, Linux (eventually), Chrome OS, Android. (Those platforms where the Chrome built-in DNS resolver is available.)

Team

ericorth@chromium.org
net-dev@chromium.org

Bug

crbug.com/1206455

Code affected

Network stack

Background

SVCB/HTTPS Records

HTTPS DNS records are a new IETF standard, currently a draft (draft-ietf-dnsop-svcb-https) but at this point stable and widely regarded to be ready for implementation. This standard defines two new DNS record types, SVCB and HTTPS, that are both designed to provide information about service endpoints hosted at a domain name in order to facilitate connections to those endpoints.

SVCB is the general version of this new record type, while HTTPS is a variant specifically designed for facilitating connections to HTTPS web services. As a web browser, Chrome's interest in these records is primarily to facilitate HTTPS connections, and therefore Chrome will primarily use the HTTPS record variant. This document will also frequently only specify "HTTPS", even when referring to SVCB and HTTPS in general, and most code implemented in Chrome for dealing with the records will be capable of dealing with either variant.

Summary of Chrome-relevant information communicated via HTTPS records:

- **HTTPS upgrade.** Mere presence of an HTTPS record (but not just SVCB records) indicates that all HTTP resources at that domain name are available via HTTPS. A well-behaved client would use this information to upgrade an HTTP request to an HTTPS request, avoiding susceptibility to many forms of SSL Stripping attacks. In this way, HTTPS records can act similarly to HSTS (HTTP Strict Transport Security). Unlike HSTS, this protection is available from the first connection to a site, without many of the hefty requirements of mechanisms like the HSTS Preload List.
- **ECH keys.** Encrypted Client Hello (ECH) is a new TLS extension to encrypt the TLS ClientHello message, especially for the purpose of encrypting the Server Name Indication (SNI), which traditionally reveals the user's visited web domain in the clear. ECH is defined by draft-ietf-tls-esni.

  HTTPS is the distribution mechanism for cryptographic keys and any other configuration information needed for ECH.
- **Domain aliasing.** DNS aliasing is traditionally done using CNAME records, but due to conflicting requirements, CNAME records are not allowed at apex domains, e.g. example.com, only subdomains, e.g. www.example.com. Sites that need to alias apex domains (especially common for CDN scenarios), need to take less-ideal steps such as HTTP-redirecting to CNAME-able subdomains or maintaining DNS mappings to point directly at the correct destination addresses.

HTTPS includes an alias form that acts similarly to CNAME but is also compatible with apex domains. Unlike CNAME, which is handled entirely by DNS recursive resolver servers, HTTPS aliasing can be resolved either by clients or servers to maximize compatibility with legacy DNS servers and maximize performance with updated DNS servers.

- **Protocol Upgrade.** HTTPS allows communicating information about supported protocols such as QUIC, along with the information necessary to perform the protocol connection. Such protocol upgrades are traditionally performed using mechanisms like Alt-Svc, which can often require a “wasted” connection before discovering the available upgrade and “starting over” with the upgraded connection. Alt-Svc upgrade information can sometimes be cached or hard-coded, but HTTPS allows an alternate and standardized mechanism to communicate available protocol upgrades for any HTTPS-serving domain from before the first connection.

Because browser usage of DNS has been stable for many years around making only A and AAAA queries, there is a general ossification concern around the risk when querying a new type of DNS record, breaking Chrome users or the DNS ecosystem. One specific concern is that, in order to avoid a theorized security issue with ECH, the HTTPS spec states that if HTTPS queries are made via a secure channel such as DoH, SERVFAIL or timed out responses should be considered fatal for the entire connection attempt. As a result of this, if some portion of the web responds to unknown queries or query types with SERVFAIL or by blackholing the query (either of which behavior would be against traditional DNS specs), the affected domains could become unusable by Chrome when enabling HTTPS.

Previous Experiments

In 2020Q4 and 2021Q1, Chrome has been running initial experiments (design doc) to query and parse both HTTPS records and a new experiment-invented record called INTEGRITY. When the experiment is active for one of these types, the record is queried, any results are parsed, metrics are recorded, and then Chrome continues its traditional behavior as normal without any effect from the HTTPS or INTEGRITY records received.

Experiment is still ongoing, but initial early results suggest that HTTPS can generally be queried and parsed successfully by Chrome, at least via DoH. Overall performance of HTTPS queries is roughly on-par with A/AAAA queries. Blackholed, SERVFAIL, and network failure queries seem maybe rare enough to reasonably allow following the spec and blocking the connection, but it is difficult to differentiate between this case and “normal” transient timed out queries. Overall, the results generally look good to push forward, but not so good that we won’t need to keep a careful eye on similar data as usage of HTTPS is rolled out.

Google employees can see more detailed data analysis notes.

In 2021Q3, Chrome began a new version of the experiment to also make the same experimental queries via Classic DNS rather than just DoH.

Akamai Research

At the DNS OARC 34 conference (2021-02), Akamai presented research regarding HTTPS queries and the ability of the DNS ecosystem to properly respond to the queries. Their research showed that DNS servers' ability to properly handle queries for the new type has improved rapidly as clients (especially Apple clients) have started making more HTTPS queries.

Also, in a review of about 12 million domains, they found only 4652 that reliably gave a "wrong" response, primarily the timeout/blackhole response of particular interest to Chrome. This very small number (0.03%) seems small enough to avoid significant worry over blocking connections to those domains, but unknown if any of these problematic domains have an outsized impact on Chrome queries. That will need to be researched further through Chrome's query experiments.

Other Browsers/Clients

As announced at WWDC in 2020-06, Safari has been querying HTTPS for almost all HTTP connections since 2020Q4. Unclear if this is browser-specific support or applying to all resolutions going through the OS on iOS (and maybe MacOS). Also unclear what behavior changes have been made based on the received HTTPS records. But overall, the success making HTTPS queries alone is a strong sign that Chrome can safely make similar queries.

Design

DNS Stack Interface

All host resolution in Chrome goes through the `net::HostResolver` interface. In general, this interface currently takes the domain to be queried as a `net::HostPortPair` and returns a `net::AddressList` (essentially an RFC-3484-ordered list of `net::IPEndPoint` objects) of address results. On resolution error, HostResolver currently merges almost all errors into a single `ERR_NAME_NOT_RESOLVED` net error and provides a `net::ResolveErrorInfo` to provide the underlying reason behind the error.

HTTPS-required error

A new error code, tentatively called `ERR_DNS_NAME_HTTPS_ONLY`, will be used to indicate that a name being attempted for an HTTP connection has a compatible HTTPS record and thus should be retried using HTTPS. As this is mostly a control-flow error, expected to be acted upon by connection code higher up the stack, this will be a top-level error, rather than the typical merged `ERR_NAME_NOT_RESOLVED`.

Adding scheme to input hostname

As HTTPS records are only of use for HTTP or HTTPS (or websocket) web requests, and the DNS stack needs to know which of those protocols is the case to determine whether or not to return `ERR_DNS_NAME_HTTPS_ONLY`, the request scheme will need to be added to the `net::HostResolver` inputs. This will be done by converting the current `net::HostPortPair` parameter into a `url::SchemeHostPort`.

In the main request flow, the `net::HostPortPair` is created from a `GURL` in `net::HttpStreamFactory::JobController::DoCreateJobs()`. The destination name is then plumbed through and saved in the socket pool until being passed to `net::HostResolver` in the socket connection logic. Most of these saved/passed `net::HostPortPair` instances will be replaced with `url::SchemeHostPort`, serving the same purpose except maintaining the scheme. If this replacement affects any fields used for keying the socket pool, this is not expected to have any significant impact on pooling because the socket pool is already bifurcated by SSL usage.

`url::SchemeHostPort` is slightly more restrictive on the data it can store compared to `net::HostPortPair` and has some minor difference quirks, and while these differences may cause some minor migration pain, it is not expected to be an insurmountable issue.

- It can only handle “standard” schemes, rather than non-standard schemes like "blob", "filesystem", "data", and "javascript" or unknown empty schemes. This should not be an issue for the main request flow because any such URL used with the socket pool or the DNS stack is expected to have a “standard” scheme. But there are also cases where there is no scheme (e.g. SOCKS proxies) or there potentially is a scheme, but that scheme is completely unknown to the network stack (e.g. for the socket API). To support such cases, we will add new methods to continue accepting `net::HostPortPair`, which would disable making HTTPS queries. To discourage (accidental) use of these methods compared to the `url::SchemeHostPort` methods (since only the with-scheme methods will be able to use the advantages of HTTPS records), instead of being pure overloads, they will use separate `CreateWithoutScheme()` methods and similar as the hostname is plumbed through the stack. When both `url::SchemeHostPort` and `net::HostPortPair` are accepted, it will be internally stored using an `absl::variant`. This should only be necessary for `net::HostResolver` and `net::ConnectJob` layers because HTTP stream factory and main socket pool logic is only used for standard HTTP/HTTPS/WS/WSS requests and can exclusively use `url::SchemeHostPort`.
- It requires canonicalized hostnames. Also expected to already be the case for anything going through the main request flow as the canonicalization is performed when initializing URLs into `GURL`s (and the `GURL`s are then used to initialize the `net::HostPortPair`). Corner cases, if not currently canonicalizing input, may need to do so as an extra step.

- It stores IPv6 literals within brackets, while `net::HostPortPair` expects the brackets to be stripped and currently does so within `net::HostPortPair::FromURL()`. Deeper in the stack, the host resolution logic attempts to resolve as IP addresses by attempting `net::IPAddress::AssignFromIPLiteral()`, which expects no brackets (and actually internally adds brackets to it in order to pass to code that expects brackets). Best solution is likely to make `net::IPAddress` more flexible and able to handle input both with and without the brackets.

Per Section 8.6 of the HTTPS RFC, it is correct to use HTTPS queries for WS/WSS URLs, not SVCB. Therefore host resolution logic will also recognize WS/WSS schemes and handle them accordingly. But this should not be necessary to handle the main request flow where, in order to prevent the socket pool from unnecessarily separating HTTP/HTTPS sockets from WS/WSS sockets, WS/WSS schemes will be massaged into HTTP/HTTPS schemes.

net::AddressList replacement

Output from the DNS stack takes the form of `Get*Results()` methods on `net::HostResolver::ResolveHostRequest`. Different query types provide data to different result methods, but because HTTPS results can provide different values for different endpoint services, the HTTPS data will need to be grouped with address results rather than being a separate result type. An update/replacement will be created for `GetAddressResults()` and its `net::AddressList` return type that will return HTTPS-derived data, grouped by endpoint service.

The new return type will be a multimap (attempting to get permission for `absl::btree_multimap` in this discussion thread, but may have to settle for `std::multimap`) that maps a new protocol/version struct (tentatively called `net::EndpointProtocol` and using `net::NextProto`) to a new data type tentatively called `net::EndpointServiceConnectionInfo`:

```
struct EndpointServiceConnectionInfo {
  std::vector<net::IPEndPoint> ipv4_endpoints;
  std::string ipv4_alias_name;
  std::vector<net::IPEndPoint> ipv6_endpoints;
  std::string ipv6_alias_name;
  EchConfigList ech_config_list;
};
```

All values with the same key will be sorted by HTTPS record weight (for `std::multimap`, this should be the order in which the entries were inserted in the multimap; unsure about `absl::btree_multimap`). Note that `EchConfigList` will likely (TBD with full ECH plans) just be an alias for a raw byte vector that gets plumbed across the stack as-is to BoringSSL to handle the parsing/interpretation, and as such, not having ECH config info can just be represented by an empty vector.

This multimap or relevant subportions of it will replace all usage of `net::AddressList` in Chrome, e.g. the `net::IPEndPoint` lists alone covers most usage of `net::AddressList`, so just those lists could be passed into code such as socket connection code that doesn't need any more.

One item of note is aliasing. `net::AddressList` currently contains `dns_aliases()` and `GetCanonicalName()` methods that allow retrieving the names used in the alias chain. This has long been considered non-ideal to be stored in `net::AddressList`, especially because it has been the only thing keeping `net::AddressList` from being converted into a simple `std::vector<net::IPEndPoint>`. See this bug. Also, it's not always accurate because while `GetCanonicalName()` assumes a single final name, because separate DNS queries could receive different CNAME records, A and AAAA results could end up with different final canonical names, and HTTPS only further complicates the potential separation. Because nothing in Chrome currently ever cares about the exact alias chain, only the final name or the unordered names used in the chain, the new struct will contain the one final name for each address family (typically the service name when HTTPS service records are involved). All other names used in alias chains (including any aliases used only for HTTPS records) will be provided in no specific order at a new get*Results() method on `net::HostResolver::ResolveHostRequest`.

DNS Stack Internals

Querying HTTPS

HTTPS queries will be made very similarly to those currently made for HTTPS experiments. That is that the logic for deciding when to make the queries will live in `net::HostResolverManager::DnsTask::DnsTask()` alongside current logic for deciding what specific DNS queries/transactions to make using the built-in resolver for a given HostResolver request.

The logic will add an HTTPS transaction to the `transactions_needed_` queue for any `net::DnsQueryType::UNSPECIFIED` request (the query type used for standard web requests that currently make A and AAAA queries) when the relevant TBD feature flags are enabled. The logic will also only add HTTPS for relevant schemes (http, https, ws, and wss). This means the scheme (passed to `net::HostResolverManager` via `url::SchemeHostPort`) will need to be plumbed into the `net::HostResolverManager::DnsTask` alongside the current hostname string parameter. Also, HTTPS transactions may (TBD based on current experiments and will likely be a separate feature or parameter flag) be added only when they will be made using DoH, but `DnsTask` already has parameters to know whether or not DoH is used for the transactions. `DnsTask` will then make its various requests to `net::DnsTransaction`, same as it does for any other query type.

HTTPS query experiments currently have special timeout logic for HTTPS queries that allow the experimental queries to be made without significant risk of the overall request blocking on a slow/blackholed query. Assuming query experiments do not discover significant timeout/blackhole issues for HTTPS compared to A/AAAA, these special timeouts will not be used beyond the experiments. HTTPS queries will be subject to the same server-fallback/timeout logic as A and AAAA queries, and the overall host resolution request will not complete until all 3 queries complete (or any 1 query fails or times out). Depending on eventual experiment results for querying HTTPS via Classic DNS, we may want to introduce a timeout for HTTPS via Classic DNS, similar to experiment times, as performance may be less predictable with those servers, and the downsides of missing a potential HTTPS record will not be as severe.

Note that there is an alternative to blocking overall host resolution on receiving results for all 3 queries: Fully handling results asynchronously as they come for different types, per RFC 8305. Section 3 and Section 5.1 of the HTTPS spec discuss optional optimizations taking advantage of RFC 8305. But such support would require very significant changes to Chrome connection logic, beyond the DNS stack. Enough work that it would not be ideal to block HTTPS support on it. Therefore, such efforts are out of scope for this design and should be considered for future followup work.

Handling HTTPS Responses

For the special case where the HTTPS query times out or results in SERVFAIL while querying via DoH, per the HTTPS spec, the desired behavior is for the entire connection attempt to fail to avoid ECH downgrade attacks. As currently written, after a `net::HostResolverManager::DnsTask` failure, `net::HostResolverManager::Job` will attempt to run any fallback tasks available via `net::HostResolverManager::Job::RunNextTask()`. This is the mechanism to do things like fallback from DoH to non-DoH or fallback from the built-in DNS resolver to the system resolver. To avoid fallback for this special case, there will be a new `fatal_error` parameter to pass from `net::HostResolverManager::DnsTask::OnTransactionComplete()`, where it can be easily determined whether or not it is the special case, through the failure flow to `net::HostResolverManager::Job::OnDnsTaskFailure()` where `RunNextTask()` would normally be called. Also, as `net::HostResolverManager::DnsTask` normally cancels immediately, triggering fallback, on any A or AAAA failure, new logic will need to be written to instead wait for any pending HTTPS requests to complete or fail if conditions make those requests eligible to trigger the special behavior, in order to see if triggering fallback is appropriate.

For any HTTPS failure not meeting the special fatal failure case, HTTPS failures will generally be ignored and treated the same as a successful request with no HTTPS results. This is different from A/AAAA failures, where any failure generally fails the `net::HostResolverManager::DnsTask` and leads to potential fallback via `net::HostResolverManager::Job::RunNextTask()`. But with HTTPS, fallback will often lead to new tasks that do not support making HTTPS queries, e.g. via the system resolver, leading to a very similar end result as if Chrome ignored the HTTPS failure.

On receiving a successful HTTPS response in `net::HostResolverManager::DnsTask::OnTransactionComplete()`, the response is sent to a `net::DnsResponseResultExtractor`, which parses the response and builds a resulting `net::HostCache::Entry` (the type used both for caching results as well as the internal-to-net::HostResolverManager generalized result type). From the work done for the HTTPS query experiments, the extractor already has all logic to parse HTTPS responses, and no necessary changes are expected there.

`net::HostCache::Entry` will need to be updated to store a generalized representation of HTTPS results. Just like the replacement of net::AddressList in results from the DNS stack interface, addresses results stored in `net::HostCache::Entry` (also currently `net::AddressList`) will also need to be replaced. To assist with merging data from multiple sources (especially A/AAAA and HTTPS), this will be stored as a map (likely `base::flat_map` for the good memory performance since this will be stored in the cache, or maybe even `base::small_map` to best handle the common case of a single service name) from service name to a new data struct tentatively called `net::HostCache::ResultServiceInfo`:

```
struct ResultServiceInfo {
  struct Service {
    std::vector<net::IPAddress> ipv4_hints;
    std::vector<net::IPAddress> ipv6_hints;
    uint16_t port;
    EchConfigList ech_config_list;
    std::vector<net::EndpointProtocol> protocols;
  };

  std::vector<net::IPAddress> ipv4;  // From A
  std::vector<net::IPAddress> ipv6;  // From AAAA
  std::vector<Service> services;  // In weighted order
};
```

Note that address hints are specific to a service entry (with a specific port, protocols, ech) but any merged-in A/AAAA results apply to all service entries for the name.

In `net::DnsResponseResultExtractor`, only records recognized as supported will be extracted into a `net::HostCache::EndpointServiceInfo`. This generally means only records with “alpn” values that Chrome recognizes and knows how to convert into the appropriate `net::NextProto` value and records without any “mandatory” fields that Chrome’s parser does not know how to parse. Note that unless the record includes a “no-default-alpn” param that http/1.1 is implicitly included as an alpn value and thus the record is compatible for the “alpn” checks. As is explicitly allowed by the HTTPS RFC, to ensure good internet compatibility and reasonable fallback availability, Chrome will reject all records for a name if all records include a “no-default-alpn” param.

Followup Queries

Unlike A/AAAA there are cases where HTTPS responses could result in the need for Chrome to make additional queries. (Note that Chrome, like most DNS clients, always relies on recursive resolvers handling CNAME and including the full chain in responses seen by Chrome, so CNAME results do not ever result in followup queries in current Chrome logic.)

If Chrome receives an HTTPS alias record, but a record for the alias target is not included in the same response, Chrome will need to make an additional round of A/AAAA/HTTPS queries for the alias target name. To avoid any issues with loops or poor performance from chains, Chrome will only make one such round of followup. If more are needed, Chrome will behave as if the HTTPS records do not exist and will use the data from the original A/AAAA records alone (ignoring any A/AAAA results from the first followup query). Note that the potential for multiple alias followups should only come from successive chain links because splitting aliases at the same level are disallowed (spec states to pick one at random if it happens).

If Chrome receives a compatible HTTPS service record with a different service name than the query name, but no address hints are given, Chrome will need to make additional A/AAAA queries (but not an HTTPS query). In theory, Chrome could attempt such followups for all compatible service records, but for simplicity/performance, Chrome will only allow one such followup (possible exception for cache-only followups) per host resolution job, choosing the highest priority service record based on protocol and weight.

`net::HostResolverManager::DnsTask::OnTransactionComplete()` will be updated to recognize the conditions requiring followup queries and to modify `net::HostResolverManager::DnsTask::transactions_needed_` to queue the followup tasks. Because followup queries will generally be made for different query names, `transactions_needed_` will also need to keep track of qnames rather than just qtypes as it does now. Care will also need to be taken to ensure `net::HostResolverManager::Job` is capable of adding more than one additional transaction to a DnsTask. It has some current design to attempt to handle this sort of change, but I don't have high expectations that it will actually work for all required cases here, especially if HTTPS queries are made via Do53, requiring Job to coordinate with the job dispatcher.

HostCache Rewrite

`net::HostCache` is currently designed around caching results of whole host resolver requests. Cache keys primarily consist of the host resolution request parameters, and cached results are the overall merged result or error. In its current design, the cache would be unsuitable for caching individual DNS requests or similar used to merge into the overall response.

HTTPS creates a need for the cache to be able to provide such intermediate results. E.g., if an HTTPS query necessitates followup A/AAAA queries for example.com, it would be extremely desirable to be able to read that from the cache if example.com were already previously queried for a similar or unrelated request. The current cache would only be able to provide overall results for a similar request, missing out on possible matches with a same-named subquery but different overall parameters, and in such cases where only a portion of results are desirable, it would be difficult to separate them out from the current merged results. Similar situations for if an HTTPS alias points to another domain that could have previously been requested (or been the alias target name for a different request name). Another trait of HTTPS that leads to problems with the current shared cache is that HTTPS is more likely to have different TTLs than A and AAAA, and the current cache just uses 1 shared TTL (the min of the merged TTLs). If e.g. a domain decides to use half the TTL for HTTPS results compared to A/AAAA, the current cache would then give the result of unnecessarily doubling the frequency of the A/AAAA queries.

While a `net::HostCache` rewrite would be good for any HTTPS query support (due to the significant increase (~50%) in subqueries and TTL issues), it won't be a blocker except for support for followup queries (due to the larger reliance on interdependency between subqueries). For initial no-followup HTTPS support, the old cache will be used, and per the current design of that cache and all interactions with it, unless an entire request can be resolved via cache, the current behavior means all subqueries will be retrieved fresh from the network, even if they could theoretically be answered from the cache.

The rewritten `net::HostCache` will be keyed by individual DNS requests, thus a standard request resulting in A, AAAA, and HTTPS queries will result in 3 cached entries (more if followup queries are required), all with their own separate TTLs.

Alias links (both HTTPS aliases and CNAMEs), even when received as chains (as is typical for CNAME where Chrome always receives the entire chain in a single DNS response) will be stored as individual link entries. Since aliasing is not specific to DNS query types, alias links will be keyed with an "any" type that can match any query type. The link-separated entries will allow Chrome to use the cache for a name that may have previously been only used as the alias of a previous query. This could be especially useful for HTTPS, e.g. if website operators start using HTTPS aliases to alias from example.com to www.example.com (or vice versa), both names to which the user or site links could reasonably lead to queries. With alias links indexed separately in the cache, Chrome would be able to thereafter serve either example.com or www.example.com directly from cache. To provide consistent behavior regarding Chrome's limits on followup queries, the cache entries will keep track of whether or not they were an HTTPS alias that required Chrome to make a followup query. Resolution logic would then disallow resolving through more than one of such links.

The resolution logic currently creates individual cache entries for each subquery (in `net::DnsResponseResultExtractor`) and merges them together (in `net::HostResolverManager::DnsTask::OnTransactionComplete()`) for storage in the cache and passing the result back up the stack. The new cache would eliminate the need to merge cache entries. Individual entries will be saved into the cache as they are created, and the entries will be passed up the stack as results collectively in a `std::vector` or similar.

When using the system resolver, Chrome receives results as a single result containing both IPv4 and IPv6 results, unlike Chrome's built-in resolver which makes separate queries for each query type (and will cache the query types separately once this `net::HostCache` rewrite work is complete). For consistency and simplicity in the cache lookup logic, these combined system resolver results will also be separated by IP family and stored in separate A and AAAA cache entries (no HTTPS entry since we never get HTTPS results from the system resolver). This will also allow cache reuse if an A-only request comes when an all-family request has been recently made and cached.

On successful fallback (e.g. successful fallback to the system resolver after an error with the built-in resolver (but not timeout or SERVFAIL for HTTPS queries that will no longer allow fallback to prevent ECH downgrade attacks)), if the success is being cached for longer than any cached errors from the failed queries, a "fallback" cache entry will be saved over the cached error with the same TTL of the success. This is necessary because DNS errors are typically cached with different TTLs from successes (and not all errors are cached at all), and it would be undesirable to attempt the failed requests again just because the cached failure expired while a cached successful fallback result is still available. Conversely, if the error causing fallback has a TTL longer than the fallback success TTL, the original cached error will be kept, allowing Chrome to immediately attempt the fallback logic for that query as long as the cached error is unexpired.

During resolution, cache lookups will be done within the Task logic (e.g. `net::HostResolverManager::DnsTask`), rather than between Tasks as a pseudo-Task as is currently done, as the query-specific caching will allow the tasks to more easily lookup the queries the task is about to make. Note though that `net::HostResolverManager::ResolveLocally()` will still have a special case cache lookup attempt to run before any Tasks are run in order to allow synchronous results if possible. The Task will also be able to make partial queries, e.g. if `net::HostResolverManager::DnsTask` finds that it has an unexpired A cache entry but not an AAAA entry, it could use the cached A while doing a fresh query just for AAAA. This is especially relevant for HTTPS where server operators are a little more likely to use different TTLs than they were for A vs AAAA. But while partial caching will be used within a Task, for logical simplicity, fallback will be all or nothing. If fallback is needed for one query (including if any one query has a cached error or cached "fallback" entry), the fallback Task will attempt to get all query types and cached values recognized by the previous Task will only be used for fallback if compatible (e.g. an insecure DnsTask could read entries cached by any DnsTask, secure or insecure).

Some slight changes will also be beneficial in the cancellation logic for `net::HostResolverManager::Job`. Currently, when the last attached `net::HostResolverManager::RequestImpl` is cancelled, the entire job and all its underlying tasks are immediately cancelled, meaning any pending results will never be cached. With all the changes to make intermediate results usable from the cache, a cancelling `net::HostResolverManager::Job` might as well wait for any pending queries to complete and cache those results. Should still cancel before making any additional requests (followup or fallback). Most of this work will be some simple additional logic in the task scheduling to recognize a cancelling job and not move on to the next task.

Network Connection Logic Changes

HTTPS connection upgrade

On returning an `ERR_DNS_NAME_HTTPS_ONLY` error result, the error will be allowed to return up the stack to `net::URLRequestHttpJob::OnStartCompleted()`. At that point, redirect headers will be synthesized to mimic an actual HTTP redirect response, following the example of the very similar logic in `net::URLRequestHttpJob::Create()` and `net::URLRequestRedirectJob` that creates a 307 redirect for HSTS-matching hostnames and then simulates receiving fake headers for that invented 307 to make use of the standard logic for following actual HTTP header redirects. The correct behavior for receiving HTTPS records is very similar as the HTTPS RFC spec defines the correct behavior to be to treat it as if receiving a 307 redirect. Note that while it would be ideal to directly reuse `net::URLRequestRedirectJob`, that would be difficult to do because DNS results are not available until after the `net::URLRequestHttpJob` has started, and logic to restart the request with a new job carries risks of novel and duplicating signals being sent to various delegates and watchers as nothing has previously ever done a restart at that stage in a `net::URLRequestJob` lifecycle.

One notable difference from HSTS however is that HSTS requires that “user recourse” (allowing users to click through various HTTPS cert errors) be disallowed. HTTPS record upgrade has no such requirement (merely allowing the behavior as a “MAY”). In conversations with Chrome security and UX experts, we believe, slightly counter-intuitively, for it to be ideal to allow user recourse, similarly to any normal HTTPS website. This is because there are a number of scenarios where it is necessary for reasons outside the website’s control to fix as well as research showing a low clickthrough rate for it being used to bypass security in non-ideal cases. I believe the desired result will be achieved in implementation as long as no special action is taken to replicate the HSTS behavior. I believe the security interstitials code receives the information from `net::SSLInfo::is_fatal_cert_error`, which is sourced from HSTS-specific code in `net::TransportSecurityState::ShouldSSLErrorsBeFatal()`.

Encrypted Client Hello (ECH)

Most of the implementation of ECH is out of scope for this document and is covered in a separate design doc. Here we concern ourselves only with plumbing ECH keys and configuration from the DNS stack to TLS code.

For TLS, as interaction between sockets and DNS results generally occurs within the socket transport connection code, e.g. see the accesses to the host resolution request object in `net::TransportConnectJob::DoTransportConnect()`, the SSL/QUIC connection code will have to retrieve the keys from the underlying transport code. So the transport connection code, on successful connection will record which `net::EndpointServiceConnectionInfo` to which it succeeded in connecting and provide an accessor to the SSL/QUIC layer to retrieve either the ECH config info for that connection or maybe the entire `net::EndpointServiceConnectionInfo`. This accessor can then be accessed by the SSL connection code in `net::SSLConnectJob`. This logic is also expected to be sufficient for HTTP/2, which uses the same ConnectJobs because the switch to HTTP/2 isn't made until after the TLS handshake.

QUIC code currently interacts with DNS in `net::QuicStreamFactory::Job` before passing the results to and making various connection-related method calls to a `net::QuicChromiumClientSession`. Should be a simple matter of passing ECH keys in one of the method calls or replacing the current `net::AddressList` params with `net::EndpointServiceConnectionInfo`.

Protocol upgrade

Alt-Svc information is currently stored (and persisted to disk) in the `net::HttpServerProperties`, itself stored in the `net::HttpNetworkSession`. In `net::HttpStreamFactory::JobController::DoCreateJobs()`, the stored Alt-Svc information is checked, and if a compatible Alt-Svc info is saved, separate "main" and "alternate" `net::HttpStreamFactory::Job`s are created, with the "main" one initially paused for some dynamically-determined delay before the two `net::HttpStreamFactory::Job`s are allowed to race.

The current pattern doesn't work well for HTTPS-based upgrade because, as DNS is available before the first connection, it is desired to use DNS results to affect that first connection, but DNS queries are not made until later during socket creation, after the point when separate `net::HttpStreamFactory::Job`s are created. It would also not be desirable to move up the DNS process to this point because extra DNS requests would be an unnecessary delay if a socket is already open in the socket pool. It would also not be desirable to store DNS-based upgrade information in `net::HttpServerProperties` as it would add considerable complexity to keep that information in sync with `net::HostCache`; especially complex would be dealing with cases where the HTTPS information changes in between reading from `net::HttpServerProperties` and checking DNS results.

Rather than just the current "main" and conditional "alternate", `net::HttpStreamFactory::JobController` will be modified to start multiple `net::HttpStreamFactory::Job`s, all coordinated by the `net::HttpStreamFactory::JobController`, without any prior knowledge of whether or not a DNS-based upgrade will be available once DNS is queried. This gives the following potential jobs (in order of preference if multiple are usable):

1. Alt-Svc (if QUIC)
2. DNS-based QUIC upgrade (if QUIC enabled and request protocol is HTTPS)

3. Alt-Svc (if not QUIC)
4. Non-upgrade ("Normal" HTTP)

(Note that only one Alt-Svc `net::HttpStreamFactory::Job` will be run per `net::HttpStreamFactory::JobController::DoCreateJobs()` call, for a maximum of 3 jobs. Also, Alt-Svc is currently disallowed for non-QUIC (because nothing ever changes `net::HttpNetworkSession::enable_http2_alternative_service` from its default `false`), so Job (3) might not exist either way. Undecided if, while making all these other changes, support will be fully added to create jobs for non-QUIC or if support will be fully added to actually allow non-QUIC Alt-Svc. TBD when it can be better determined how much extra work it would take.)

Alt-Svc (QUIC or non-QUIC) and the non-upgrade `net::HttpStreamFactory::Job`s will behave about the same as the current "alternate" and "main" `net::HttpStreamFactory::Job`s respectively. TODO: Decide if any cleanup work will be done to split up `net::HttpStreamFactory::Job`, e.g. into separate classes by quic vs non-quic.

The DNS-based QUIC upgrade `net::HttpStreamFactory::Job` will behave similarly to Alt-Svc for QUIC upgrade, except when the `net::HttpStreamFactory::Job` calls into `net::QuicStreamFactory`, it will provide a new parameter (name TBD) to specify that `net::QuicStreamFactory` should fail if, after performing host resolution (the normal resolution done as part of current `net::QuicStreamFactory` code), the DNS results do not specify an upgrade to QUIC. The new parameter will likely need to be added to the `net::QuicSessionKey` to keep `net::QuicStreamFactory::Job`s from being merged for different post-DNS lookup requirements. See below for a mitigating optimization to occur if redundant QUIC `net::HttpStreamFactory::Job`s are present after no already-connected sockets/streams are found.

Each of these `net::HttpStreamFactory::Job`s will, in priority order, check a new `net::ClientSocketPoolManager` function to request a socket if and only if a compatible socket is already available and connected (or for QUIC, similar functionality added in `net::QuicStreamFactory` to determine if a compatible QUIC session is already available). If any upgrade `net::HttpStreamFactory::Job`s (Alt-Svc or DNS-based) find an already-connected socket/session, it will be used and all other `net::HttpStreamFactory::Job`s are cancelled (in the case of higher priority `net::HttpStreamFactory::Job`s that did not have a connected socket/session) or never run (in the case of lower-priority `net::HttpStreamFactory::Job`s yet to begin the check). After checking for an already-connected socket/session, each non-cancelled `net::HttpStreamFactory::Job` will wait for a signal from the `net::HttpStreamFactory::JobController` before proceeding further. Even if Alt-Svc QUIC and DNS-based `net::HttpStreamFactory::Job`s use the same destination, both must check for already-available QUIC session because both `net::HttpStreamFactory::Job`s would result in different session keys, and it is possible that a DNS-based net::HttpStreamFactory::Job previously created a session before the current mostly-redundant Alt-Svc configuration was set.

For Alt-Svc upgrade, a non-upgrade socket has typically connected first, so it is desirable to ignore it for a while and attempt the upgrade net::HttpStreamFactory::Job. To accomplish this, iff an Alt-Svc entry was found but no already-connected QUIC stream was available for the upgrade, a non-upgrade net::HttpStreamFactory::Job that finds an already-connected socket will not cause the Alt-Svc net::HttpStreamFactory::Job to be cancelled. Both net::HttpStreamFactory::Jobs will proceed to the next step and race.

DNS-based upgrade has no reliance on a non-upgrade socket ever being previously connected, so if one is available, that is a strong signal that previous DNS-based upgrade attempts did not succeed in upgrade (because there was no upgrade available in DNS results or because the attempts failed or were slow). Therefore if a non-upgrade net::HttpStreamFactory::Job finds an already-connected socket (which will only be checked if the DNS-based upgrade did not find an already-connected QUIC stream), the DNS-based upgrade net::HttpStreamFactory::Job will be canceled as a performance optimization to avoid blocking the request on the DNS queries necessary to attempt DNS-based upgrade. As an additional optimization, if both Alt-Svc QUIC upgrade and DNS-based QUIC upgrade net::HttpStreamFactory::Jobs are uncancelled, if they are for the same destination parameters (mostly the case if Alt-Svc does not change the destination host or port), they are redundant (both expected to give the same result except one will potentially stop after DNS), so the DNS-based upgrade net::HttpStreamFactory::Job will be canceled at that time.

If no net::HttpStreamFactory::Job finds an already-connected socket (or only a non-upgrade net::HttpStreamFactory::Job does but there is still an Alt-Svc net::HttpStreamFactory::Job to race it with), the non-cancelled net::HttpStreamFactory::Jobs will race to attempt connection. Each net::HttpStreamFactory::Job will begin working further in priority order with a short delay between starting each net::HttpStreamFactory::Job. This will all be coordinated by the net::HttpStreamFactory::JobController. The amount of delay will be based on the current logic for delaying the non-upgrade net::HttpStreamFactory::Job behind Alt-Svc net::HttpStreamFactory::Jobs, which dynamically determines a suitable delay for non-QUIC net::HttpStreamFactory::Jobs based recent QUIC connection performance, generally at least 300ms and at most 3 seconds. This delay will allow higher-priority net::HttpStreamFactory::Jobs to be preferred, even if lower-priority net::HttpStreamFactory::Jobs are able to connect quicker (or are already connected). Note that resulting DNS lookups for all these net::HttpStreamFactory::Jobs will often be redundant, but net::HostResolver should generally be able to re-use the same results, either through merging redundant net::HostResolver::ResolveHostRequests or retrieving previous results from cache, except in very rare cases where the cache expires in the short time between DNS lookups.

Just like with the current “main” and “alternate” logic, once any `net::HttpStreamFactory::Job` successfully completes, all other still-running or waiting `net::HttpStreamFactory::Job`s are cancelled by the `net::HttpStreamFactory::JobController`. And once any `net::HttpStreamFactory::Job` completes unsuccessfully, the next-priority `net::HttpStreamFactory::Job`, if not yet started, will be started without delay.

TODO: Consider if there’s further optimization room by more intelligently merging together `net::QuicStreamFactory::Job`s for the same destination, differing only by the new parameter specifying if it should fail without a DNS-based upgrade path. Could be important for cases around abnormal timing delays in various concurrent `net::HttpsStreamFactory::JobController`s, resulting in concurrent Alt-Svc and DNS-based `net::HttpStreamFactory::Job`s that will not cancel each other on first completion (maybe because another concurrent `net::HttpsStreamFactory::JobController` succeeded via a non-upgrade `net::HttpStreamFactory::Job` and found Alt-Svc data that wasn’t available for a still-running DNS-based `net::HttpStreamFactory::Job`). Maybe on success (or earlier), a `net::QuicStreamFactory::Job` should look for `net::QuicStreamFactory::Job`s of the other type and cancel/merge with them. But care must be taken to ensure `net::HttpStreamFactory::Job`s cannot result in QUIC connections when there is no QUIC upgrade path currently available (e.g. because Alt-Svc info expired and Chrome no longer has any relevant Alt-Svc info). In current Chrome logic, once Alt-Svc info is gone, no “alternate” `net::HttpStreamFactory::Job` will run, and thus QUIC will not be used, even if a compatible QUIC session is available. To maintain that behavior, a DNS-based upgrade `net::HttpStreamFactory::Job` cannot be allowed to connect to that session, at least not without first confirming with DNS results that a DNS-based upgrade is available to the same server. But those dangers mostly only apply to whether or not to allow a DNS-based `net::HttpStreamFactory::Job` to merge with an Alt-Svc job/session. Once unexpired Alt-Svc info is found and and Alt-Svc `net::HttpStreamFactory::Job` is created, it should be safe for it to merge with any DNS-based `net::QuicStreamFactory::Job`s that have already passed the DNS stage.

Metrics

Success metrics

TODO: Success is mostly just that this gets used (HTTPS records received, HTTP requests upgraded, protocol upgrades, ECH keys received), so add the obvious metrics for that usage. But it may be a long-term thing after all implementation is complete before we really see a lot of that and can declare “success”.

Regression metrics

TODO: Mostly just the typical mix of DNS-specific and Chrome-wide performance metrics. Make sure we don't slow anything down.

New metrics:

- Various HTTP record cases (e.g. incompatible "alpn", incompatible "mandatory", etc)
- The security-sensitive cases (timeout or SERVFAIL over DoH)
- HTTPS-record-specific performance/results (mostly just adapted from the current query experiment metrics)

Experiments

Most behavior covered by a new "UseDnsHttpsSvcb" Finch Feature. The Feature itself will control whether or not HTTPS will be queried alongside A/AAAA queries. Additional Finch FeatureParams:

- "UseDnsHttpsSvcbHttpUpgrade" controls whether or not HostResolver will emit the `ERR_DNS_NAME_HTTPS_ONLY` error on receiving HTTPS results for http-schemed requests. It will also be checked by the redirect logic expected to handle the error.
- "UseDnsHttpsSvcbEnforceSecureResponse" controls whether or not Chrome uses the special handling for HTTPS timeout or SERVFAIL responses received via DoH to treat such failures as fatal to the entire request. This behavior is the behavior desired to prevent ECH downgrade attacks.
- "UseDnsHttpsSvcbEnableInsecure" controls whether HTTPS queries will be made via Classic DNS or only when DoH is in use.
- "UseDnsHttpsSvcbExtraTimeAbsolute" and "UseDnsHttpsSvcbExtraTimePercent" are used to control extra timeout behavior specific to HTTPS transactions still running after A and AAAA transactions have completed. This timeout behavior mirrors the timeouts created for the initial query-only experiments.

ECH will primarily be controlled through separate experiments. If HTTPS is queried (controlled by the "UseDnsHttpsSvcb" Feature) and ECH config information is received, `net::HostResolver` will always output it with results. The connection and BoringSSL code will then use its own experiments to control whether or not that information is used to protect the connections with ECH.

TODO: TBD Finch control for connection protocol upgrade behavior.

Rollout plan

The first planned launch of HTTPS-based behavior is the HTTP->HTTPS upgrade functionality, and only for simpler cases where the HTTPS record does not redirect to other DNS names.

This will be launched via the “UseDnsHttpsSvcb” Feature with the “UseDnsHttpsSvcbHttpUpgrade” FeatureParam. TBD based on other ongoing experiments and initial rollout metrics how “UseDnsHttpsSvcbEnableInsecure”, “UseDnsHttpsSvcbExtraTimeAbsolute”, and “UseDnsHttpsSvcbExtraTimePercent” will be set for this launch.

TODO: Plan subsequent launches. Likely ECH behavior, then handling for more advanced records with aliasing and followup queries and such, then handling for protocol upgrade.

Core principle considerations

Speed

Expect performance improvements for websites that support HTTPS-based QUIC upgrade (those with an HTTPS record that lists the QUIC protocol). Should make QUIC upgrade (and the resulting performance improvements) a little more common. Should make QUIC upgrade itself a little more performant by eliminating unnecessary connection roundtrips.

Some potential negative implications due to waiting on the DNS resolves themselves (because host resolution blocks until the slowest of A, AAAA, and HTTPS completes), but with all the DNS caching (both in Chrome and DNS servers) HTTPS should have about the same performance as A/AAAA and it is generally expected to be a mostly insignificant impact. Biggest risk is that HTTPS will often be negative results while A/AAAA are often positive, and it is hypothesized that negative results may be very slightly slower overall in DNS. Will carefully monitor during rollouts to ensure no major impact.

Simplicity

Mostly expected to be invisible-to-user technical changes. Users shouldn’t notice anything other than maybe that they end up on https:// pages a little more often and that Chrome describes their websites as “insecure” a little less often. Performance and security improvements without any user action.

No direct user configurability is planned. The only expected reason a user/enterprise would ever want to disable HTTPS queries would be if it breaks something in a network (which should not happen unless the network is doing something buggy and extremely counter to many old and foundational DNS standards). Users/enterprises will be able to disable HTTPS queries via DoH by disabling DoH (currently supported through group policy or via Chrome’s “Secure DNS” configuration UI). If Chrome supports HTTPS queries though non-DoH (TBD), a temporary group policy will be provided to allow enterprises to disable the queries until their network bugs are fixed.

Security

Multiple security improvements expected:

- The DNS-based HTTP->HTTPS connection upgrade subfeature should be able to prevent some forms of SSL stripping attacks. Unless an attacker is also able to block/alter the DNS results, websites are able to signal to Chrome that HTTPS is expected to be functional and Chrome will disallow connecting to the site via HTTP. An attacker could not successfully force HTTP by blocking the HTTP site redirecting to HTTPS or force a fallback to HTTP by blocking access to HTTPS. This is very similar to the protection provided by HSTS. Except note that the “no user recourse” features of HSTS are not reflected here; a DNS-based HTTPS upgrade will allow the same normal-Chrome-behavior user recourse as if the user had just typed in an https:// URL or received an HTTP redirect to one.
- The fact that HTTPS records and their various features only work for sites that support HTTPS connections may lead to some small portion of websites better supporting HTTPS connections in order to access desired features of the records (e.g. the better DNS aliasing).
- ECH config retrieval should unblock the security/privacy improvements of ECH.
- Improved access to QUIC could improve security of some websites and Chrome’s connections to them.

No expected increases of vulnerabilities. Like any other DNS mechanisms, Chrome will receive the information without any direct validation that the information comes from the domain name owner. If using DoH, the information is at-best only as trustworthy as the user’s trust in their DNS server, and if DoH is not used, there is even less protection. But because the information from HTTPS records is only used for HTTPS connections, standard TLS encryption and validation should protect users from any new attacks other than attempting to manipulate/block HTTPS records to block a user from being able to access a site. But it is assumed that an attacker that can manipulate/block HTTPS records would be able to do the same for A/AAAA records for the same result. E.g., an attacker could attempt to insert false HTTPS alias records redirecting to an attacker-controlled server, but unless the attacker server has access to certs for the origin name, Chrome would refuse to connect to the false server and the user is no worse off than if the attacker had manipulated the A/AAAA records. No change is ever made to the origin except the

HTTP->HTTPS redirect, which is limited to only ever allowing the scheme portion of the origin to change from “http” to “https” (or “ws” to “wss”).

To further clarify the vulnerability if HTTPS queries are made via Classic DNS: If an attacker blocks HTTPS responses, leading to false negatives, the user gets no advantages from HTTPS and is generally no worse off than if Chrome did not support HTTPS. At worst, an attacker could attempt to slow Chrome’s performance by delaying HTTPS responses and making Chrome wait additional time or by designing HTTPS responses that Chrome will work extra hard to parse. But overall, this isn’t significantly different from the capabilities of a similar attacker doing the same manipulations on A or AAAA responses. Conversely, if an attacker manipulated HTTPS responses to provide false data the damage is limited from the fact that the data is only used to assist with making https:// connections that must always have a TLS cert matching the origin from before any HTTPS manipulation. At worst, an attacker could prevent Chrome from successfully connecting to a website, or divert the connection attempt to an unaffiliated server that can never pass TLS cert checks and will just receive extra useless connection attempts from Chrome. This is not significantly different from the capabilities of a similar attacker modifying A or AAAA responses.

Of special note is that an on-path attacker (either between Chrome and the recursive resolver or between the recursive resolver and other resolvers) could attempt to block/manipulate HTTPS records to force a downgrade attack from HTTPS-related features to a normal A/AAAA-only connection, e.g. preventing the use of ECH. There is no protection from such attacks unless the user is using DoH. If using DoH, an attacker would have to be sophisticated enough to identify and block the HTTPS packets while allowing the A/AAAA packets through normally. Chrome’s protection from such attacks will be to disallow any connection to a site on receiving a timed out response (indicating a potential attack between Chrome and the DNS server) or a SERVFAIL response (indicating a potential attack between the DNS recursive resolver and other resolver servers) to an HTTPS DoH query. This will ensure a worst-case attack of blocking access to a site (which we assume any similarly capable attackers could do with or without HTTPS records) rather than forcing a security-downgraded connection to the site. This protection will be implemented before implementing support for ECH.

Privacy considerations

No privacy implications expected other than the improved security/privacy of the resulting site connections. Making HTTPS queries (alongside A/AAAA queries) doesn’t reveal anything about Chrome or the user that wouldn't be revealed by the A/AAAA queries except that Chrome is a new enough version to support such queries.

Testing plan

Unit tests as well as adding basic HTTPS query tests to the DoH-related integration tests: `HttpWithDnsOverHttpsTest` and `DohBrowserTest` (note that HTTPS queries cannot be exercised in most general browser tests because most of the DNS stack is generally bypassed in Chrome browser tests). Fuzzers already in place to cover the DNS record parsing code and the main code (`net::HostResolverManager`) for handling DNS query/response behavior.

Followup work

- If performance doesn't meet expectations, consider implementing Happy Eyeballs v2 (RFC 8305) to better deal with performance differences between multiple DNS query types.
- Implement full ECH support in BoringSSL and Chrome's TLS connection logic to take advantage of the ECH configs received through HTTPS records. Design doc.