Tracing HTTP requests (nettrace)

Motivation

- In EVE, HTTP is the main application protocol used to carry the management traffic as well as for downloading images
    - Used for all communication between device and controller
    - Used to verify network connectivity (i.e. network connectivity errors often come from Golang's HTTP client)
    - Used to download application and EVE images (including for AWS, Azure and GCP storages) - the only exception is SFTP datastore (not sure how frequently used by customers)
- Troubleshooting of customer-reported network issues is quite challenging …
- EVE only reports the final error as returned by HTTP client - typically wrapping multiple errors as returning back up the stack, but often important information from lower layers is lost
- Difficult to retrospectively backtrace to the root cause
- Single HTTP request hides a complicated sequence of operations behind the scenes, possibly consisting of:
    - multiple DNS requests (A + AAAA, multiple DNS servers)
    - additional TCP connection attempts (if multiple resolved IPs are available and some fail)
    - reused TCP connections (previously opened)
    - TLS handshakes (some possibly with reused sessions), server cert verification
    - HTTP/HTTPS proxying
    - HTTP redirects
- When we receive error message, it is often difficult to determine which underlying operation has triggered it and what has led to it - for example, each of the operations has some timeout and when we receive "Context deadline exceeded" it is hard or impossible to tell which operation has failed to finalize in time or consumed unexpected amount of it
- Image downloading is even more difficult because we do not know (without very close code inspection) what a 3rd party library (like github.com/aws/aws-sdk-go) is doing behind the scenes (potentially running multiple HTTP requests) - we often get errors like "<*very-very-long-url*>: connection timeout" - how did we get there and what is this particular request doing? Download process gets even more convoluted if customer has put a load-balancer in front of the datastore, e.g. Azure Traffic Manager.
- When we cannot progress with a customer issue, we often ask the customer to make some observations and collect information for us, such as interface packet counters, conntrack entries, packet trace, etc., by invoking curl/ping/dig command on our behalf
    - would be more efficient (and professional) if we could get such information automatically alongside a logged/published network error
- Same error can have different causes - for example "Context deadline exceeded" may be returned when TCP SYN packet has no response (possibly blocked by a firewall) as well as when larger packets are not getting through (e.g. MTU issue). Without some connection stats (like bytes sent, received) it is difficult to tell them apart.

Nettrace

In order to improve observability of the HTTP request processing (in EVE and possibly is other projects written in Go as well), a new Golang package nettrace was developed. The idea was to hook into Golang's HTTP client and record a summary of all important operations performed at different network layers of the HTTP request processing, which, when combined, provide a clear picture of what was happening and in what order behind the scenes. When a request fails, it should be much easier to backtrace to its root cause. We decided to call this process "***network tracing***" and the output "***network trace***" (taking some inspiration from the trace package).

After some experimentation and analysis of the Golang's implementation of the HTTP client, we were able to find several places in the code at different layers of the network stack that we can safely hook into. Safely here means that it does not affect or limit the client's functionality or the caller's expectations. Also, we put an extra effort on making sure that tracing is implemented effectively, without introducing too many extra cycles or any blocking calls, so that the performance of HTTP request processing is not noticeably affected (some overhead cannot be avoided).

We were able to for example:

1. Wrap http.Transport (implementing RoundTripper interface) and observe HTTP requests/responses at the application layer. Note that some code in EVE and in 3rd party libs used by zedUpload assumes that RoundTripper's concrete type is http.Transport. Inside EVE we simply changed such code to remove this assumption and in the 3rd party libraries this is not reachable from EVE, because we pass a ready to use `http.Client` so that a 3rd party library does not need to do any further customization. In any case, there is an option to disable this hook (at the cost of losing tracing info on the application and the session (TLS) layers - see point 2.). On the other hand, it will not cause any issues if the caller is adding some additional Transport wrapping on top of it (like we do here).
2. With Transport wrapped, we can add ClientTrace into the request context to call our callbacks and inform us about: TCP connection selected, TLS handshake(s) done/failed (incl. server certs and negotiated parameters) and some more (less useful info). Note that it will work even if the caller also adds its own ClientTrace (like EVE does here), because httptrace.WithClientTrace is smart enough to be able to merge multiple traces.
3. Put a custom DialContext (into the underlying http.Transport) as well as a custom Dial for Resolver, both of which still call the original methods, but also wrap the returned net.Conn to observe socket read/write operations.
4. In our custom Dial methods, use Dialer.Control to get access to information about every TCP connection attempt (including failed ones).

These hooks allow us to collect information from the user-space, i.e. from the application down to the socket layer (TCP/UDP payload), but not further below. However, since EVE is already using connection tracking provided by netfilter, we can also collect conntrack entries for every recorded TCP connection and UDP "exchange" (DNS). This provides us with information like:

- Packet, byte counters on the L3 layer
- How was the connection NATed (mapping between address 5-tuples)
- Connection mark - aka EVE firewall rule ID applied to the connection
- TCP states (SYN_SENT, ESTABLISHED, …)
- Conntrack's own connection states (UNREPLIED, ASSURED, …)

And finally, there is the packet capture, which we secretly always wish to have included with every network-related customer issue. We provide an option to capture all packets of the traffic produced during an HTTP request processing (including e.g. nameserver queries). Packets are captured from the kernel using BPF, parsed using gopacket package and filtered out to contain only those that correspond to a traced HTTP request. This, of course, is a costly operation affecting network performance, so the idea is to do it only on special occasions, with small HTTP requests (not image download) and infrequently (and allow to disable it altogether) - for a more detailed description of when this is used in EVE, see Dumping network diagnostics.

The scope of traced and collected information is configurable. For example, it is possible to enable/disable retrieval of conntrack entries.

When we collect and combine all this information, we are able to report:

- All Dial attempts. Each record contains:
  - reference to trace record of established TCP connection (undefined if failed)
  - dial begin + end time, context close time
  - destination address
  - proxy config
  - static source IP (if set, otherwise undefined)
  - dial error (nil if OK)
- All DNS queries. Each record contains
  - reference to trace record of Dial where this originated from
  - reference to trace record of the underlying UDP or TCP connection (used as a fallback from truncated UDP DNS response)
  - (optional) sent DNS questions and received DNS message header + answers (we are able to parse DNS messages from sent/received data)
- All TCP connections (attempts + established). Each record contains:
  - reference to trace record of Dial where this originated from
  - handshake start + done time, conn close time
  - 4-tuple (src IP, src port, dst IP, dst port)
  - was it reused?
  - total sent + received bytes (L4 payload)
  - (optional) conntrack (captured-at time, 5-tuple after NAT, mark, flags, packet/byte counters)
  - (optional) socket trace - array of:
    - operation type (read or write), op begin+end time, transferred data length, error (nil if OK)
- All UDP "connections" (or rather exchanges of messages). Each record contains:
  - reference to trace record of Dial where this originated from
  - time when the socket was created and when it was closed
  - 4-tuple (src IP, src port, dst IP, dst port)
  - total sent + received bytes (L4 payload)
  - (optional) conntrack (captured-at time, 5-tuple after NAT, mark, flags, packet/byte counters)
  - (optional) socket trace - array of:
    - operation type (read or write), op begin+end time, transferred data length, error (nil if OK)
- All TLS tunnels (attempted + established). Each record contains:
  - reference to trace record of the underlying TCP connection
  - was resumed from a previous session?
  - handshake start + done time, error (nil if OK)
  - negotiated cipher and application proto
  - SNI value
  - for every peer cert in the chain:
    - subject, issuer, validity time range (NotBefore, NotAfter)
- All HTTP requests made. Each record contains info for both the request and the response:
  - reference to trace record of the underlying TCP connection
  - reference to trace record(s) of the underlying TLS tunnel(s) (2 tunnels are made with proxy listening on HTTPS)
  - time when the request was sent
  - method, URL, HTTP version
  - (optional) request headers
  - request message content length (not transport length which can differ)
  - time when response was received, error (nil if OK)
  - response status code, HTTP version
  - (optional) response headers
  - response message content length (not transport length which can differ)

“optional” means that the particular feature is configurable and provides an option to disable it.

The above is defined and stored inside a Go structure **HTTPTrace** with json tags (so that it can be stored as a nice readable JSON file).

Packet captures, if enabled, are returned separately (one per single selected interface) and can be stored into .pcap files.

For a more detailed documentation and the usage of the nettrace package, please refer to the README file.

HTTP tracing inside EVE

We decided to collect and publish HTTP traces from three microservices: downloader, nim and zedagent.

Downloader

Downloader is used to download EVE and application images. This depends on 3rd party libraries, all of which (except one) are using an HTTP client to download all image parts from a given datastore. The process to download a single image can be quite complicated, potentially involving parallel HTTP requests retrieving different parts of an image. Being able to read a recording of events that led to a failed download is therefore a prime use-case for the HTTP tracing.

By default, this does not include any packet captures. However, a limited PCAP can be enabled (with config option `netdump.downloader.with.pcap`). Limited in the sense that it will not include TCP segments carrying non-empty payload (i.e. packets with the downloaded data and as it happens also TLS handshakes). On the other hand, included will be all UDP traffic (DNS requests), TCP SYN, RST and FIN packets as well as ACK packets without data piggybacking (sufficient to tell where a download process got "broken"). The total PCAP size is limited to 64MB (packets past this limit will not be included).

Always disabled is tracing of socket operations (reads and writes), which would make the recording large and hard to read. Still, even without socket trace and packet capture (by default), obtained HTTP trace allows us to learn the sequence and timing of all HTTP requests made, DNS queries executed, all TCP connections opened and used, the outcome of TLS handshakes and packet/bytes statistics.

Since image downloads are not very frequent processes, it makes sense to always enable tracing (of the limited scope as mentioned above). Downloader always publishes obtained trace, regardless of the outcome of the download process. Even a trace of a successful download can be useful, e.g. for comparison purposes or simply to learn how and where did EVE download all the image parts from.

NIM

It is very useful to at least sporadically trace communication between the device and the controller, especially when EVE detects a connectivity issue. There are multiple microservices communicating with the controller: client, zedagent, nim, diag. The last mentioned was created specifically to monitor and probe device connectivity status and report it on the console output. This would seem as a fitting place for some HTTP tracing. However, the problem is that diag is not in control of the device network configuration (aka DPC) - instead, nim is. As a result, we often see a spurious failure reported by diag caused by nim switching from one network configuration to another while diag was running a connectivity test. And in fact, if the latest network config is evaluated by nim as not working (i.e. not providing working connectivity with the controller), nim will immediately fallback to a previous DPC (if any is available), leaving very small window for other microservices to even attempt to use the latest DPC.

Note that nim is evaluating connectivity status by running a `/ping` request towards the controller. If the request fails, the error returned from `http.Client` is reported inside DevicePortStatus. With traced `http.Client`, we are able to obtain and publish more information when the connectivity checks are failing. Additionally, during a traced connectivity check, a failed `/ping` request is followed by GET requests for `http://www.google.com` and `https://www.google.com` (just like diag is performing). From trace output we are then therefore able to tell if, at the time of the test, the device had any Internet connectivity at all.

As opposed to the downloader, in nim it makes sense to include all tracing information, including packet capture so that we can narrow down the root cause of a failed check as much as possible. However, we should then perform tracing much less frequently - not with each connectivity check performed by nim, which is at least once every 5 minutes. Multiple traces obtained inside a duration of the same network issue would likely not add any additional information. We decided to run full HTTP tracing only at most once per hour before onboarding and at most once per day after onboarding (the second interval is configurable, learn more here) and only when the latest DPC is being tested. It does not make sense to troubleshoot obsolete network configurations.

Just like downloader, nim publishes traces of both successful and failed connectivity checks.

Zedagent

With nim sporadically tracing `/ping` and `google.com` requests, it still makes sense to utilize network tracing in zedagent as well. This microservice is running the most important requests: `/config` to get the latest device configuration, aka the intended state, and `/info` to publish the actual device state. In fact, both of these must succeed for the device to be considered as Online and not as Suspect by zedcloud. As it was pointed out above, a failing latest DPC is applied only temporarily - until nim performs one connectivity check and fallbacks to a previous DPC. This means that as long as the latest DPC is marked as not working, it does not make sense for zedagent to trace its requests, because they would likely be using an obsolete DPC anyway. However, if nim evaluates the latest DPC as working yet zedagent is failing to get config or publish information (specifically ZInfoDevice), then zedagent is eligible to run tracing and publish the output. However, the same tracing interval (at most once per hour/day before/after onboarding by default) applies here as well.

Both successful and failed config/info network traces are published.