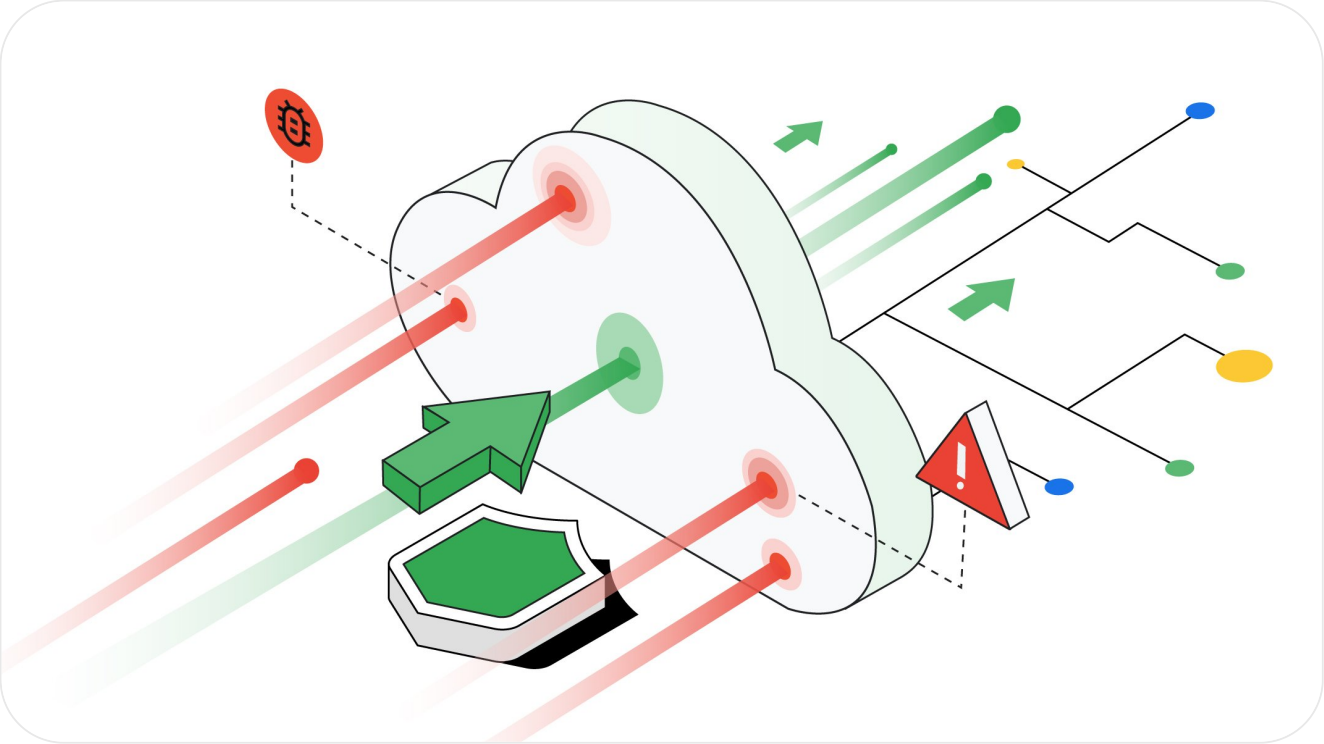
Security & Identity

How it works: The novel HTTP/2 'Rapid Reset' DDoS attack

October 10, 2023



Daniele Iamartino

Juho Snellman

A number of Google services and Cloud customers have been targeted with a novel HTTP/2-based DDoS attack which peaked in August. These attacks were significantly larger than any previously-reported Layer 7 attacks, with the largest attack surpassing 398 million requests per second.

The attacks were largely stopped at the edge of our network by Google's global load balancing infrastructure and did not lead to any outages. While the impact was minimal, Google's DDoS Response Team reviewed the attacks and added additional protections to further mitigate similar attacks. In addition to Google's internal response, we helped lead a coordinated disclosure process with industry partners to address the new HTTP/2 vector across the ecosystem.

Below, we explain the predominant methodology for Layer 7 attacks over the last few years, what changed in these new attacks to make them so much larger, and the mitigation strategies we believe are effective against this attack type. This article is written from the perspective of a reverse proxy architecture, where the HTTP request is terminated by a reverse proxy that forwards requests to other services. The same concepts apply to HTTP servers that are integrated into the application server, but with slightly different considerations which potentially lead to different mitigation strategies.

A primer on HTTP/2 for DDoS

Since late 2021, the majority of Layer 7 DDoS attacks we've observed across Google first-party services and Google Cloud projects protected by Cloud Armor have been based on HTTP/2, both by number of attacks and by peak request rates.

efficient for legitimate clients can also be used to make DDoS attacks more efficient.

Stream multiplexing

HTTP/2 uses "streams", bidirectional abstractions used to transmit various messages, or "frames", between the endpoints. “Stream multiplexing” is the core HTTP/2 feature which allows higher utilization of each TCP connection. Streams are multiplexed in a way that can be tracked by both sides of the connection while only using one Layer 4 connection. Stream multiplexing enables clients to have multiple in-flight requests without managing multiple individual connections.

One of the main constraints when mounting a Layer 7 DoS attack is the number of concurrent transport connections. Each connection carries a cost, including operating system memory for socket records and buffers, CPU time for the TLS handshake, as well as each connection needing a unique four-tuple, the IP address and port pair for each side of the connection, constraining the number of concurrent connections between two IP addresses.

In HTTP/1.1, each request is processed serially. The server will read a request, process it, write a response, and only then read and process the next request. In practice, this means that the rate of

includes the network latency, proxy processing time and backend request processing time. While HTTP/1.1 pipelining is available in some clients and servers to increase a connection's throughput, it is not prevalent amongst legitimate clients.

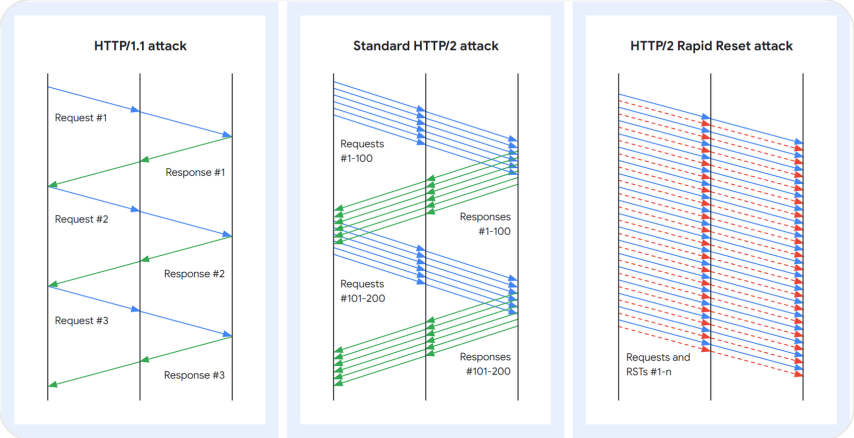
With HTTP/2, the client can open multiple concurrent streams on a single TCP connection, each stream corresponding to one HTTP request. The maximum number of concurrent open streams is, in theory, controllable by the server, but in practice clients may open 100 streams per request and the servers process these requests in parallel. It's important to note that server limits can not be unilaterally adjusted.

For example, the client can open 100 streams and send a request on each of them in a single round trip; the proxy will read and process each stream serially, but the requests to the backend servers can again be parallelized. The client can then open new streams as it receives responses to the previous ones. This gives an effective throughput for a single connection of 100 requests per round trip, with similar round trip timing constants to HTTP/1.1 requests. This will typically lead to almost 100 times higher utilization of each connection.

The HTTP/2 Rapid Reset attack

sending a RST_STREAM frame. The protocol does not require the client and server to coordinate the cancellation in any way, the client may do it unilaterally. The client may also assume that the cancellation will take effect immediately when the server receives the RST_STREAM frame, before any other data from that TCP connection is processed.

This attack is called Rapid Reset because it relies on the ability for an endpoint to send a RST_STREAM frame immediately after sending a request frame, which makes the other endpoint start working and then rapidly resets the request. The request is canceled, but leaves the HTTP/2 connection open.



HTTP/1.1 and HTTP/2 request and response pattern

The HTTP/2 Rapid Reset attack built on this capability is simple: The client opens a large number of streams at once as in the standard HTTP/2 attack, but rather than waiting for a response to each request stream from the server or proxy, the client cancels each request immediately.

in flight. By explicitly canceling the requests, the attacker never exceeds the limit on the number of concurrent open streams. The number of in-flight requests is no longer dependent on the round-trip time (RTT), but only on the available network bandwidth.

In a typical HTTP/2 server implementation, the server will still have to do significant amounts of work for canceled requests, such as allocating new stream data structures, parsing the query and doing header decompression, and mapping the URL to a resource. For reverse proxy implementations, the request may be proxied to the backend server before the RST_STREAM frame is processed. The client on the other hand paid almost no costs for sending the requests. This creates an exploitable cost asymmetry between the server and the client.

Another advantage the attacker gains is that the explicit cancellation of requests immediately after creation means that a reverse proxy server won't send a response to any of the requests. Canceling the requests before a response is written reduces downlink (server/proxy to attacker) bandwidth.

HTTP/2 Rapid Reset attack variants

In the weeks after the initial DDoS attacks, we have seen some Rapid Reset attack variants. These

standard HTTP/2 DDoS attacks.

The first variant does not immediately cancel the streams, but instead opens a batch of streams at once, waits for some time, and then cancels those streams and then immediately opens another large batch of new streams. This attack may bypass mitigations that are based on just the rate of inbound RST_STREAM frames (such as allow at most 100 RST_STREAMs per second on a connection before closing it).

These attacks lose the main advantage of the canceling attacks by not maximizing connection utilization, but still have some implementation efficiencies over standard HTTP/2 DDoS attacks. But this variant does mean that any mitigation based on rate-limiting stream cancellations should set fairly strict limits to be effective.

The second variant does away with canceling streams entirely, and instead optimistically tries to open more concurrent streams than the server advertised. The benefit of this approach over the standard HTTP/2 DDoS attack is that the client can keep the request pipeline full at all times, and eliminate client-proxy RTT as a bottleneck. It can also eliminate the proxy-server RTT as a bottleneck if the request is to a resource that the HTTP/2 server responds to immediately.

RFC 9113, the current HTTP/2 RFC, suggests that an attempt to open too many streams should invalidate

servers will not process those streams, and is what enables the non-cancelling attack variant by almost immediately accepting and processing a new stream after responding to a previous stream.

A multifaceted approach to mitigations

We don't expect that simply blocking individual requests is a viable mitigation against this class of attacks — instead the entire TCP connection needs to be closed when abuse is detected. HTTP/2 provides built-in support for closing connections, using the GOAWAY frame type. The RFC defines a process for gracefully closing a connection that involves first sending an informational GOAWAY that does not set a limit on opening new streams, and one round trip later sending another that forbids opening additional streams.

However, this graceful GOAWAY process is usually not implemented in a way which is robust against malicious clients. This form of mitigation leaves the connection vulnerable to Rapid Reset attacks for too long, and should not be used for building mitigations as it does not stop the inbound requests. Instead, the GOAWAY should be set up to limit stream creation immediately.

This leaves the question of deciding which connections are abusive. The client canceling

request processing. Typical situations are when a browser no longer needs a resource it had requested due to the user navigating away from the page, or applications using a long polling approach with a client-side timeout.

Mitigations for this attack vector can take multiple forms, but mostly center around tracking connection statistics and using various signals and business logic to determine how useful each connection is. For example, if a connection has more than 100 requests with more than 50% of the given requests canceled, it could be a candidate for a mitigation response. The magnitude and type of response depends on the risk to each platform, but responses can range from forceful GOAWAY frames as discussed before to closing the TCP connection immediately.

To mitigate against the non-cancelling variant of this attack, we recommend that HTTP/2 servers should close connections that exceed the concurrent stream limit. This can be either immediately or after some small number of repeat offenses.

Applicability to other protocols

We do not believe these attack methods translate directly to HTTP/3 (QUIC) due to protocol

Despite that, our recommendation is for HTTP/3 server implementations to proactively implement mechanisms to limit the amount of work done by a single transport connection, similar to the HTTP/2 mitigations discussed above.

Industry coordination

Early in our DDoS Response Team's investigation and in coordination with industry partners, it was apparent that this new attack type could have a broad impact on any entity offering the HTTP/2 protocol for their services. Google helped lead a coordinated vulnerability disclosure process taking advantage of a pre-existing coordinated vulnerability disclosure group, which has been used for a number of other efforts in the past.

During the disclosure process, the team focused on notifying large-scale implementers of HTTP/2 including infrastructure companies and server software providers. The goal of these prior notifications was to develop and prepare mitigations for a coordinated release. In the past, this approach has enabled widespread protections to be enabled for service providers or available via software updates for many packages and solutions.

During the coordinated disclosure process, we reserved CVE-2023-44487 to track fixes to the various HTTP/2 implementations.

The novel attacks discussed in this post can have significant impact on services of any scale. All providers who have HTTP/2 services should assess their exposure to this issue. Software patches and updates for common web servers and programming languages may be available to apply now or in the near future. We recommend applying those fixes as soon as possible.

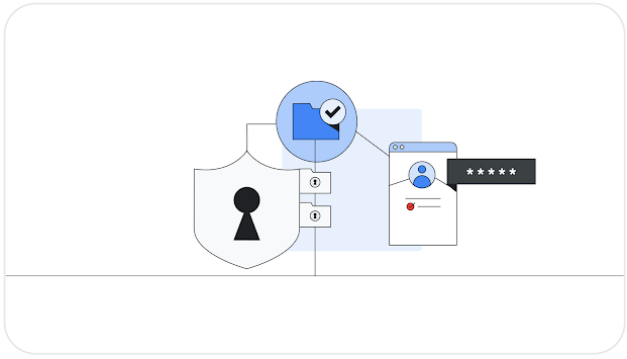
For our customers, we recommend patching software and enabling the Application Load Balancer and Google Cloud Armor, which has been protecting Google and existing Google Cloud Application Load Balancing users.

Posted in Security & Identity—Networking—Google Cloud

Related articles

Security & Identity

Cloud CISO Perspectives: Why ISACs are valuable security



Security & Identity

What's new in Assured Workloads: Japan regions, move

HTTP/2 Rapid Reset: deconstructing the record-breaking attack

10/10/2023

 Lucas Pardue  Julien Desgats

16 min read

This post is also available in 简体中文, 繁體中文, 日本語, 한국어, Deutsch, Français and Español.

Starting on Aug 25, 2023, we started to notice some unusually big HTTP attacks hitting many of our customers. These attacks were detected and mitigated by our automated DDoS system. It was not long however, before they started to reach

record breaking sizes — and eventually peaked just above 201 million requests per second. This was nearly 3x bigger than our previous biggest attack on record.

Under attack or need additional protection? Click here to get help.

Concerning is the fact that the attacker was able to generate such an attack with a botnet of merely 20,000 machines. There are botnets today that are made up of hundreds of thousands or millions of machines. Given that the entire web typically sees only between 1–3 billion requests per second, it's not inconceivable that using this method could focus an entire web's worth of requests on a small number of targets.

Detecting and Mitigating

This was a novel attack vector at an unprecedented scale, but Cloudflare's existing protections were largely able to absorb the brunt of the attacks. While initially we saw some impact to customer traffic — affecting roughly 1% of requests during the initial wave of attacks — today we've been able to refine our mitigation methods to stop the attack for any Cloudflare customer without it impacting our systems.

We noticed these attacks at the same time two other major industry players — Google and AWS — were seeing the same. We worked to harden Cloudflare's systems to ensure that, today, all our customers are protected from this new DDoS attack method without any customer impact. We've also participated with Google and AWS in a coordinated disclosure of the attack to impacted vendors and critical infrastructure providers.

This attack was made possible by abusing some features of the HTTP/2 protocol and server implementation details (see CVE-2023-44487 for details). Because the attack abuses an underlying weakness in the HTTP/2 protocol, we believe any

vendor that has implemented HTTP/2 will be subject to the attack. This included every modern web server. We, along with Google and AWS, have disclosed the attack method to web server vendors who we expect will implement patches. In the meantime, the best defense is using a DDoS mitigation service like Cloudflare's in front of any web-facing web or API server.

This post dives into the details of the HTTP/2 protocol, the feature that attackers exploited to generate these massive attacks, and the mitigation strategies we took to ensure all our customers are protected. Our hope is that by publishing these details other impacted web servers and services will have the information they need to implement mitigation strategies. And, moreover, the HTTP/2 protocol standards team, as well as teams working on future web standards, can better design them to prevent such attacks.

RST attack details

HTTP is the application protocol that powers the Web. HTTP Semantics are common to all versions of HTTP — the overall architecture, terminology, and protocol aspects such as request and response messages, methods, status codes, header and trailer fields, message content, and much more. Each individual HTTP version defines how semantics are transformed into a "wire format" for exchange over the Internet. For example, a client has to serialize a request message into binary data and send it, then the server parses that back into a message it can process.

HTTP/1.1 uses a textual form of serialization. Request and response messages are exchanged as a stream of ASCII characters, sent over a reliable transport layer like TCP, using the following format (where CRLF means carriage-return and linefeed):

```
HTTP-message   = start-line CRLF
                 *( field-line CRLF )
```

```
CRLF
[ message-body ]
```

For example, a very simple GET request for `https://blog.cloudflare.com/` would look like this on the wire:

```
GET / HTTP/1.1 CRLFHost: blog.cloudflare.comCRLFCRLF
```

And the response would look like:

```
HTTP/1.1 200 OK CRLFServer: cloudflareCRLFContent-Length:
100CRLFtext/html; charset=UTF-8CRLFCRLF<100 bytes of data>
```

This format **frames** messages on the wire, meaning that it is possible to use a single TCP connection to exchange multiple requests and responses. However, the format requires that each message is sent whole. Furthermore, in order to correctly correlate requests with responses, strict ordering is required; meaning that messages are exchanged serially and can not be multiplexed. Two GET requests, for `https://blog.cloudflare.com/` and `https://blog.cloudflare.com/page/2/`, would be:

```
GET / HTTP/1.1 CRLFHost: blog.cloudflare.comCRLFCRLFGET /page/2/ HTTP/1.1
CRLFHost: blog.cloudflare.comCRLFCRLF
```

With the responses:

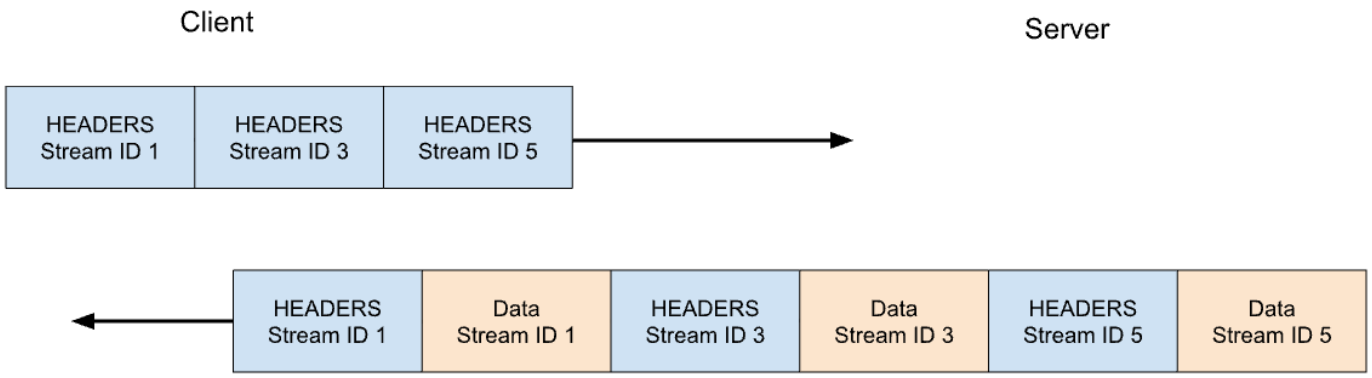
```
HTTP/1.1 200 OK CRLFServer: cloudflareCRLFContent-Length:
100CRLFtext/html; charset=UTF-8CRLFCRLF<100 bytes of data>CRLFHTTP/1.1
200 OK CRLFServer: cloudflareCRLFContent-Length: 100CRLFtext/html;
charset=UTF-8CRLFCRLF<100 bytes of data>
```

Web pages require more complicated HTTP interactions than these examples. When visiting the Cloudflare blog, your browser will load multiple scripts, styles and media assets. If you visit the front page using HTTP/1.1 and decide quickly to

navigate to page 2, your browser can pick from two options. Either wait for all of the queued up responses for the page that you no longer want before page 2 can even start, or cancel in-flight requests by closing the TCP connection and opening a new connection. Neither of these is very practical. Browsers tend to work around these limitations by managing a pool of TCP connections (up to 6 per host) and implementing complex request dispatch logic over the pool.

HTTP/2 addresses many of the issues with HTTP/1.1. Each HTTP message is serialized into a set of **HTTP/2 frames** that have type, length, flags, stream identifier (ID) and payload. The stream ID makes it clear which bytes on the wire apply to which message, allowing safe multiplexing and concurrency. Streams are bidirectional. Clients send frames and servers reply with frames using the same ID.

In HTTP/2 our GET request for `https://blog.cloudflare.com` would be exchanged across stream ID 1, with the client sending one HEADERS frame, and the server responding with one HEADERS frame, followed by one or more DATA frames. Client requests always use odd-numbered stream IDs, so subsequent requests would use stream ID 3, 5, and so on. Responses can be served in any order, and frames from different streams can be interleaved.



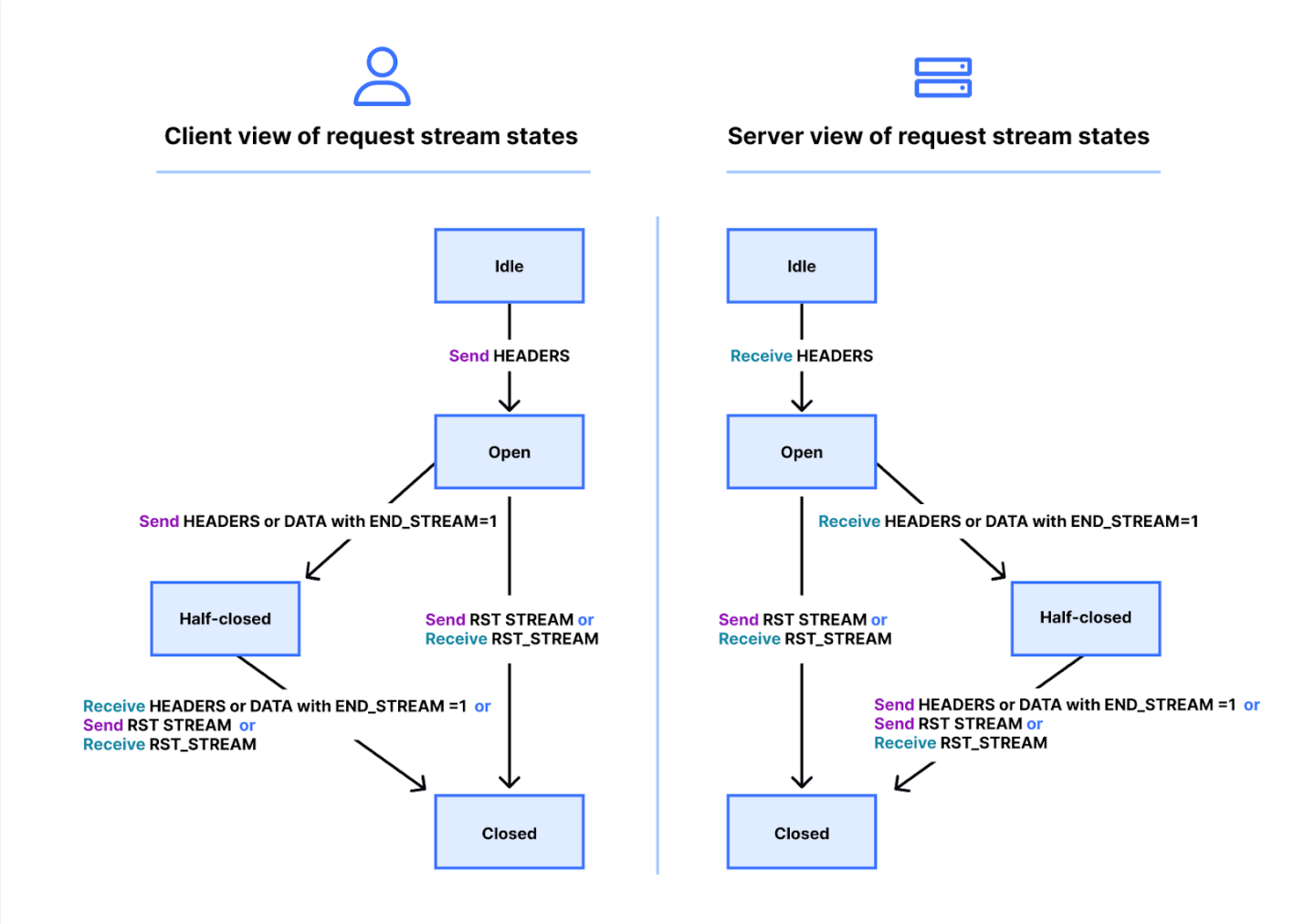
Stream multiplexing and concurrency are powerful features of HTTP/2. They enable more efficient usage of a single TCP connection. HTTP/2 optimizes resources fetching especially when coupled with prioritization. On the flip side,

making it easy for clients to launch large amounts of parallel work can increase the peak demand for server resources when compared to HTTP/1.1. This is an obvious vector for denial-of-service.

In order to provide some guardrails, HTTP/2 provides a notion of maximum active concurrent streams. The SETTINGS_MAX_CONCURRENT_STREAMS parameter allows a server to advertise its limit of concurrency. For example, if the server states a limit of 100, then only 100 requests can be active at any time. If a client attempts to open a stream above this limit, it must be rejected by the server using a RST_STREAM frame. Stream rejection does not affect the other in-flight streams on the connection.

The true story is a little more complicated. Streams have a lifecycle. Below is a diagram of the HTTP/2 stream state machine. Client and server manage their own views of the state of a stream. HEADERS, DATA and RST_STREAM frames trigger transitions when they are sent or received. Although the views of the stream state are independent, they are synchronized.

HEADERS and DATA frames include an END_STREAM flag, that when set to the value 1 (true), can trigger a state transition.



Let's work through this with an example of a GET request that has no message content. The client sends the request as a HEADERS frame with the END_STREAM flag set to 1. The client first transitions the stream from **idle** to **open** state, then immediately transitions into **half-closed** state. The client half-closed state means that it can no longer send HEADERS or DATA, only WINDOW_UPDATE, PRIORITY or RST_STREAM frames. It can receive any frame however.

Once the server receives and parses the HEADERS frame, it transitions the stream state from idle to open and then half-closed, so it matches the client. The server half-closed state means it can send any frame but receive only WINDOW_UPDATE, PRIORITY or RST_STREAM frames.

The response to the GET contains message content, so the server sends

HEADERS with END_STREAM flag set to 0, then DATA with END_STREAM flag set to 1. The DATA frame triggers the transition of the stream from **half-closed** to **closed** on the server. When the client receives it, it also transitions to closed. Once a stream is closed, no frames can be sent or received.

Applying this lifecycle back into the context of concurrency, HTTP/2 states:

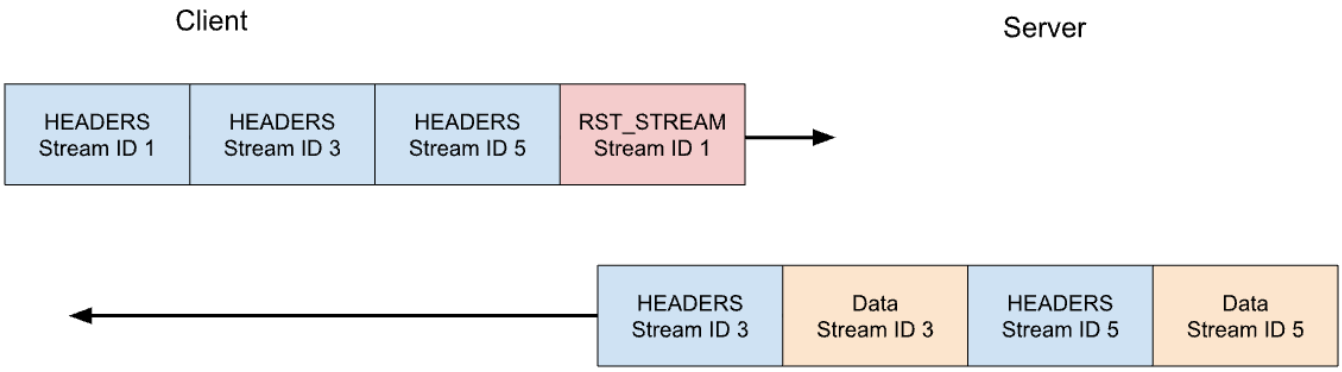
Streams that are in the "open" state or in either of the "half-closed" states count toward the maximum number of streams that an endpoint is permitted to open. Streams in any of these three states count toward the limit advertised in the SETTINGS_MAX_CONCURRENT_STREAMS setting.

In theory, the concurrency limit is useful. However, there are practical factors that hamper its effectiveness— which we will cover later in the blog.

HTTP/2 request cancellation

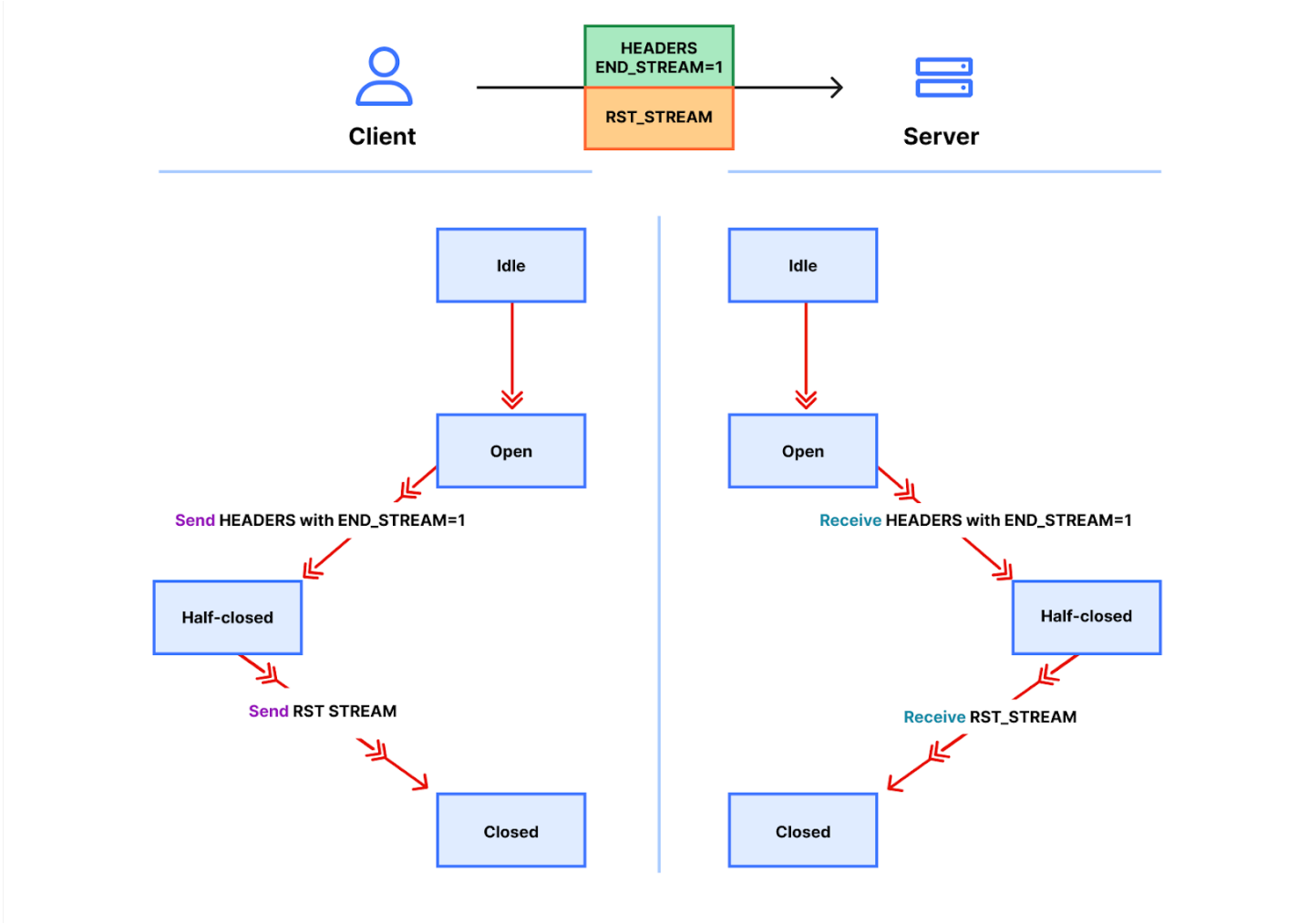
Earlier, we talked about client cancellation of in-flight requests. HTTP/2 supports this in a much more efficient way than HTTP/1.1. Rather than needing to tear down the whole connection, a client can send a RST_STREAM frame for a single stream. This instructs the server to stop processing the request and to abort the response, which frees up server resources and avoids wasting bandwidth.

Let's consider our previous example of 3 requests. This time the client cancels the request on stream 1 after all of the HEADERS have been sent. The server parses this RST_STREAM frame before it is ready to serve the response and instead only responds to stream 3 and 5:



Request cancellation is a useful feature. For example, when scrolling a webpage with multiple images, a web browser can cancel images that fall outside the viewport, meaning that images entering it can load faster. HTTP/2 makes this behaviour a lot more efficient compared to HTTP/1.1.

A request stream that is canceled, rapidly transitions through the stream lifecycle. The client's HEADERS with END_STREAM flag set to 1 transitions the state from **idle** to **open** to **half-closed**, then RST_STREAM immediately causes a transition from **half-closed** to **closed.**



Recall that only streams that are in the open or half-closed state contribute to the stream concurrency limit. When a client cancels a stream, it instantly gets the ability to open another stream in its place and can send another request immediately. This is the crux of what makes CVE-2023-44487 work.

Rapid resets leading to denial of service

HTTP/2 request cancellation can be abused to rapidly reset an unbounded number of streams. When an HTTP/2 server is able to process client-sent RST_STREAM frames and tear down state quickly enough, such rapid resets do not cause a problem. Where issues start to crop up is when there is any kind of delay or lag in tidying up. The client can churn through so many requests that a backlog of work accumulates, resulting in excess consumption of resources on the server.

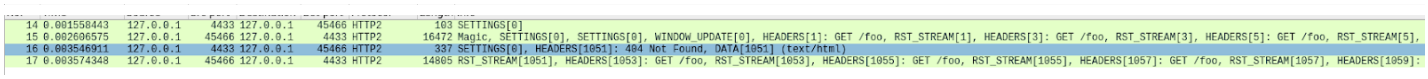
A common HTTP deployment architecture is to run an HTTP/2 proxy or load-balancer in front of other components. When a client request arrives it is quickly dispatched and the actual work is done as an asynchronous activity somewhere else. This allows the proxy to handle client traffic very efficiently. However, this separation of concerns can make it hard for the proxy to tidy up the in-process jobs. Therefore, these deployments are more likely to encounter issues from rapid resets.

When Cloudflare's reverse proxies process incoming HTTP/2 client traffic, they copy the data from the connection's socket into a buffer and process that buffered data in order. As each request is read (HEADERS and DATA frames) it is dispatched to an upstream service. When RST_STREAM frames are read, the local state for the request is torn down and the upstream is notified that the request has been canceled. Rinse and repeat until the entire buffer is consumed. However this logic can be abused: when a malicious client started sending an enormous chain of requests and resets at the start of a connection, our servers would eagerly read them all and create stress on the upstream servers to the point of being unable to process any new incoming request.

Something that is important to highlight is that stream concurrency on its own cannot mitigate rapid reset. The client can churn requests to create high request rates no matter the server's chosen value of SETTINGS_MAX_CONCURRENT_STREAMS.

Rapid Reset dissected

Here's an example of rapid reset reproduced using a proof-of-concept client attempting to make a total of 1000 requests. I've used an off-the-shelf server without any mitigations; listening on port 443 in a test environment. The traffic is dissected using Wireshark and filtered to show only HTTP/2 traffic for clarity. Download the pcap to follow along.



It's a bit difficult to see, because there are a lot of frames. We can get a quick summary via Wireshark's Statistics > HTTP2 tool:

Topic / Item	Cour
HTTP2	2008
Type	2008
HEADERS	1001
RST_STREAM	1000
SETTINGS	4
WINDOW_UPDATE	1
GOAWAY	1
DATA	1

The first frame in this trace, in packet 14, is the server's SETTINGS frame, which advertises a maximum stream concurrency of 100. In packet 15, the client sends a few control frames and then starts making requests that are rapidly reset. The first HEADERS frame is 26 bytes long, all subsequent HEADERS are only 9 bytes. This size difference is due to a compression technology called HPACK. In total, packet 15 contains 525 requests, going up to stream 1051.

```
▾ HyperText Transfer Protocol 2
  ▾ Stream: HEADERS, Stream ID: 1, Length 26, GET /foo
      Length: 26
      Type: HEADERS (1)
    ▸ Flags: 0x05, End Headers, End Stream
      0... .... .... .... .... .... .... .... = Reserved: 0x0
      .000 0000 0000 0000 0000 0000 0000 0001 = Stream Identifier: 1
      [Pad Length: 0]
      Header Block Fragment: 048362539f87418a089d5c0b8170dc69a659827a852f91d35d05
      [Header Length: 112]
      [Header Count: 5]
    ▸ Header: :path: /foo
    ▸ Header: :scheme: https
    ▸ Header: :authority: 127.0.0.1:4433
    ▸ Header: :method: GET
    ▸ Header: user-agent: example
      [Full request URI: https://127.0.0.1:4433/foo]
▾ HyperText Transfer Protocol 2
  ▾ Stream: RST_STREAM, Stream ID: 1, Length 4
      Length: 4
      Type: RST_STREAM (3)
    ▸ Flags: 0x00
      0... .... .... .... .... .... .... .... = Reserved: 0x0
      .000 0000 0000 0000 0000 0000 0000 0001 = Stream Identifier: 1
      Error: CANCEL (8)
▾ HyperText Transfer Protocol 2
  ▾ Stream: HEADERS, Stream ID: 3, Length 9, GET /foo
      Length: 9
      Type: HEADERS (1)
    ▸ Flags: 0x05, End Headers, End Stream
      0... .... .... .... .... .... .... .... = Reserved: 0x0
      .000 0000 0000 0000 0000 0000 0000 0011 = Stream Identifier: 3
      [Pad Length: 0]
      Header Block Fragment: 048362539f87bf82be
      [Header Length: 112]
      [Header Count: 5]
    ▸ Header: :path: /foo
    ▸ Header: :scheme: https
    ▸ Header: :authority: 127.0.0.1:4433
    ▸ Header: :method: GET
    ▸ Header: user-agent: example
      [Full request URI: https://127.0.0.1:4433/foo]
```

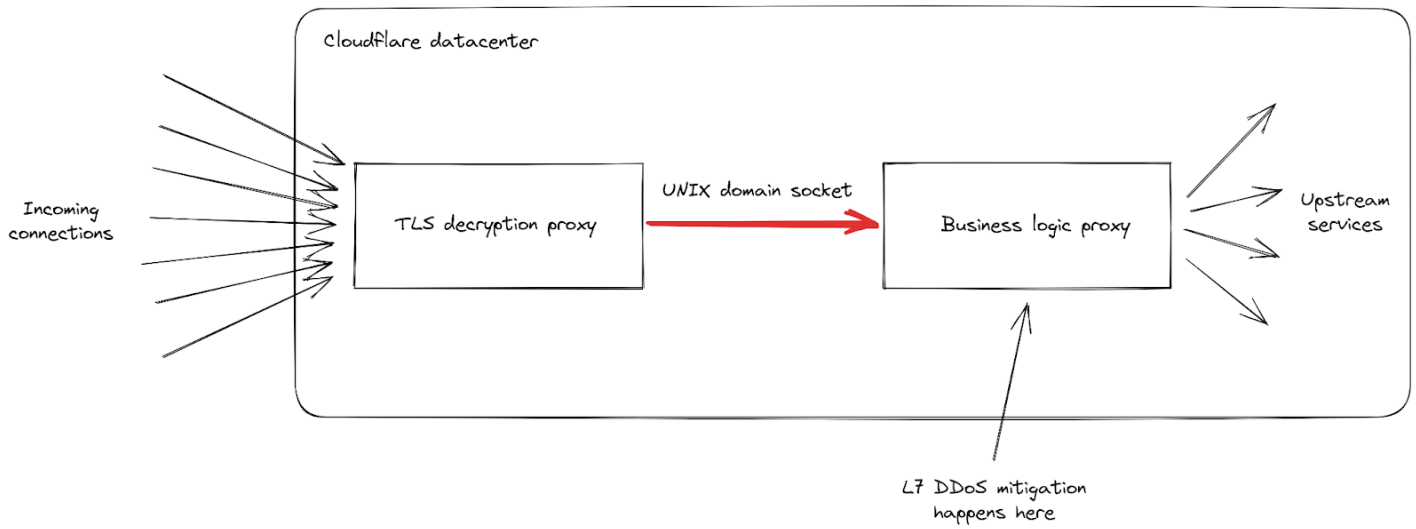
Interestingly, the RST_STREAM for stream 1051 doesn't fit in packet 15, so in packet 16 we see the server respond with a 404 response. Then in packet 17 the client does send the RST_STREAM, before moving on to sending the remaining 475 requests.

Note that although the server advertised 100 concurrent streams, both packets sent by the client sent a lot more HEADERS frames than that. The client did not have to wait for any return traffic from the server, it was only limited by the size of the packets it could send. No server RST_STREAM frames are seen in this trace, indicating that the server did not observe a concurrent stream violation.

Impact on customers

As mentioned above, as requests are canceled, upstream services are notified and can abort requests before wasting too many resources on it. This was the case with this attack, where most malicious requests were never forwarded to the origin servers. However, the sheer size of these attacks did cause some impact.

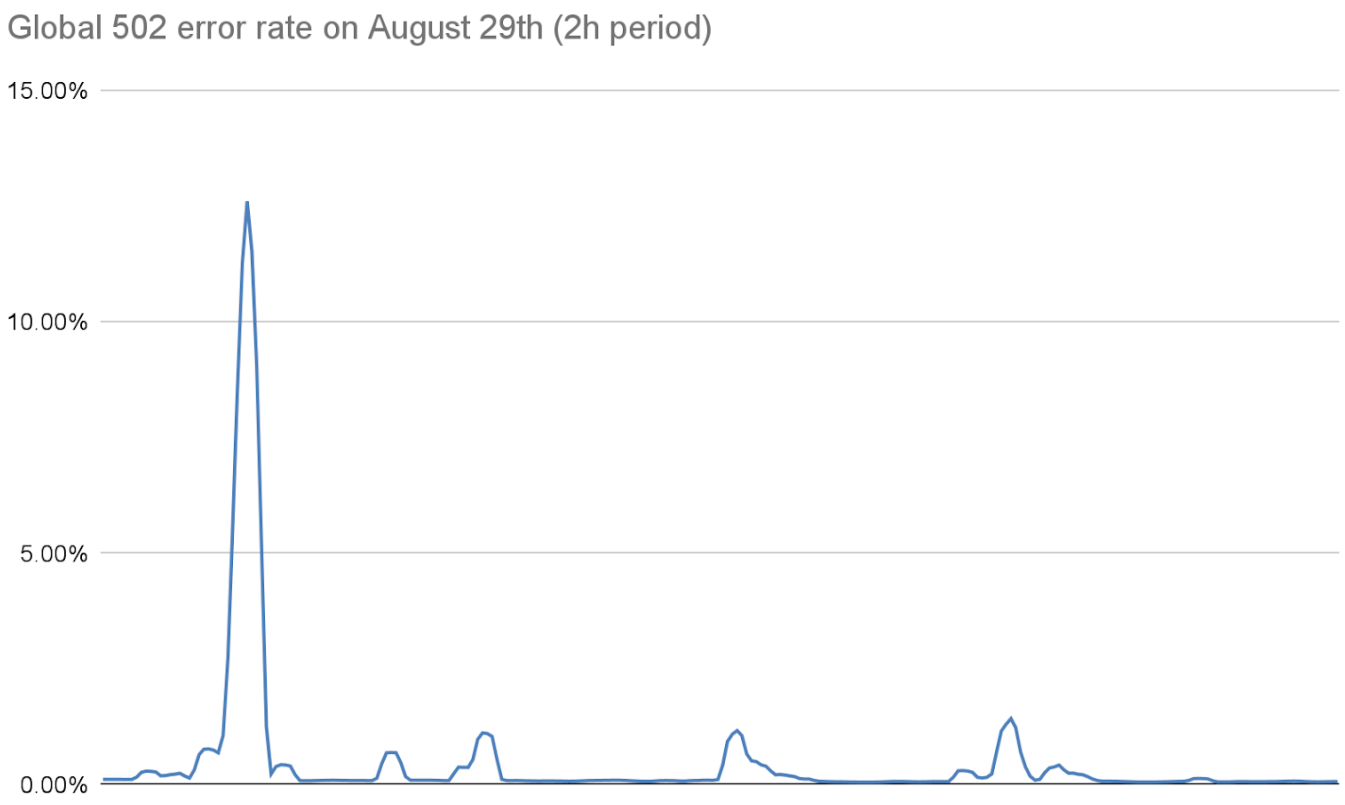
First, as the rate of incoming requests reached peaks never seen before, we had reports of increased levels of 502 errors seen by clients. This happened on our most impacted data centers as they were struggling to process all the requests. While our network is meant to deal with large attacks, this particular vulnerability exposed a weakness in our infrastructure. Let's dig a little deeper into the details, focusing on how incoming requests are handled when they hit one of our data centers:



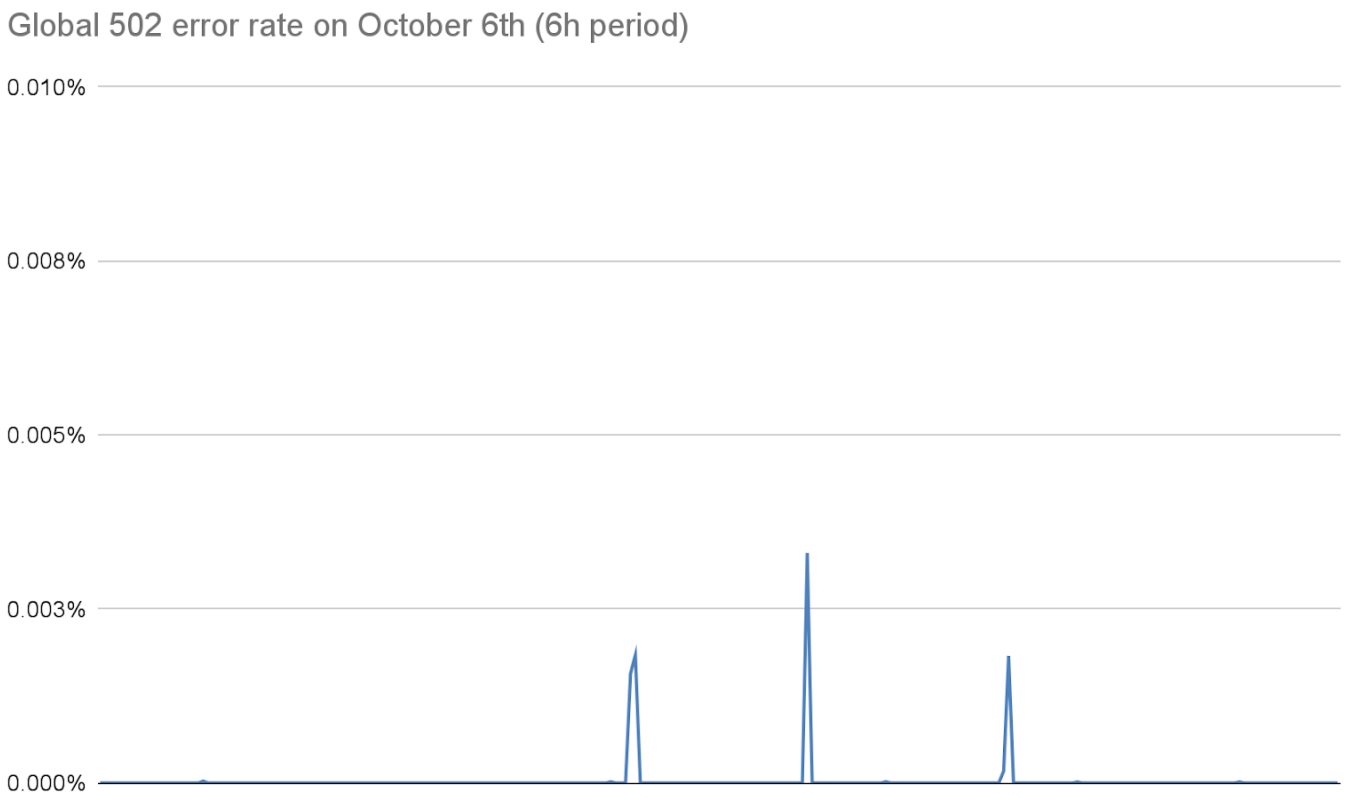
We can see that our infrastructure is composed of a chain of different proxy servers with different responsibilities. In particular, when a client connects to Cloudflare to send HTTPS traffic, it first hits our TLS decryption proxy: it decrypts TLS traffic, processes HTTP 1, 2 or 3 traffic, then forwards it to our "business logic" proxy. This one is responsible for loading all the settings for each customer, then routing the requests correctly to other upstream services — and more importantly in our case, **it is also responsible for security features**. This is where L7 attack mitigation is processed.

The problem with this attack vector is that it manages to send a lot of requests very quickly in every single connection. Each of them had to be forwarded to the business logic proxy before we had a chance to block it. As the request throughput became higher than our proxy capacity, the pipe connecting these two services reached its saturation level in some of our servers.

When this happens, the TLS proxy cannot connect anymore to its upstream proxy, this is why some clients saw a bare "502 Bad Gateway" error during the most serious attacks. It is important to note that, as of today, the logs used to create HTTP analytics are also emitted by our business logic proxy. The consequence of that is that these errors are not visible in the Cloudflare dashboard. Our internal dashboards show that about 1% of requests were impacted during the initial wave of attacks (before we implemented mitigations), with peaks at around 12% for a few seconds during the most serious one on August 29th. The following graph shows the ratio of these errors over a two hours while this was happening:



We worked to reduce this number dramatically in the following days, as detailed later on in this post. Both thanks to changes in our stack and to our mitigation that reduce the size of these attacks considerably, this number today is effectively zero.

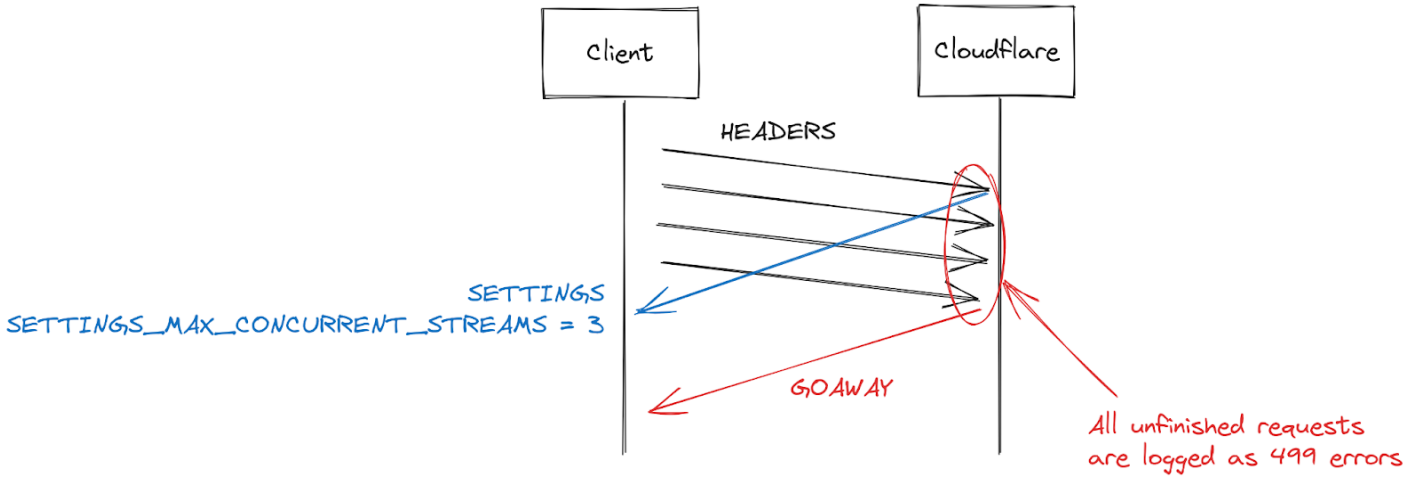


499 errors and the challenges for HTTP/2 stream concurrency

Another symptom reported by some customers is an increase in 499 errors. The reason for this is a bit different and is related to the maximum stream concurrency in a HTTP/2 connection detailed earlier in this post.

HTTP/2 settings are exchanged at the start of a connection using SETTINGS frames. In the absence of receiving an explicit parameter, default values apply. Once a client establishes an HTTP/2 connection, it can wait for a server's SETTINGS (slow) or it can assume the default values and start making requests (fast). For SETTINGS_MAX_CONCURRENT_STREAMS, the default is effectively

unlimited (stream IDs use a 31-bit number space, and requests use odd numbers, so the actual limit is 1073741824). The specification recommends that a server offer no fewer than 100 streams. Clients are generally biased towards speed, so don't tend to wait for server settings, which creates a bit of a race condition. Clients are taking a gamble on what limit the server might pick; if they pick wrong the request will be rejected and will have to be retried. Gambling on 1073741824 streams is a bit silly. Instead, a lot of clients decide to limit themselves to issuing 100 concurrent streams, with the hope that servers followed the specification recommendation. Where servers pick something below 100, this client gamble fails and streams are reset.



There are many reasons a server might reset a stream beyond concurrency limit overstepping. HTTP/2 is strict and requires a stream to be closed when there are parsing or logic errors. In 2019, Cloudflare developed several mitigations in response to HTTP/2 DoS vulnerabilities. Several of those vulnerabilities were caused by a client misbehaving, leading the server to reset a stream. A very effective strategy to clamp down on such clients is to count the number of server resets during a connection, and when that exceeds some threshold value, close the connection with a GOAWAY frame. Legitimate clients might make one or two mistakes in a connection and that is acceptable. A client that makes too many mistakes is probably either broken or malicious and closing the connection addresses both cases.

While responding to DoS attacks enabled by CVE-2023-44487, Cloudflare reduced maximum stream concurrency to 64. Before making this change, we were unaware that clients don't wait for SETTINGS and instead assume a concurrency of 100. Some web pages, such as an image gallery, do indeed cause a browser to send 100 requests immediately at the start of a connection. Unfortunately, the 36 streams above our limit all needed to be reset, which triggered our counting mitigations. This meant that we closed connections on legitimate clients, leading to a complete page load failure. As soon as we realized this interoperability issue, we changed the maximum stream concurrency to 100.

Actions from the Cloudflare side

In 2019 several DoS vulnerabilities were uncovered related to implementations of HTTP/2. Cloudflare developed and deployed a series of detections and mitigations in response. CVE-2023-44487 is a different manifestation of HTTP/2 vulnerability. However, to mitigate it we were able to extend the existing protections to monitor client-sent RST_STREAM frames and close connections when they are being used for abuse. Legitimate client uses for RST_STREAM are unaffected.

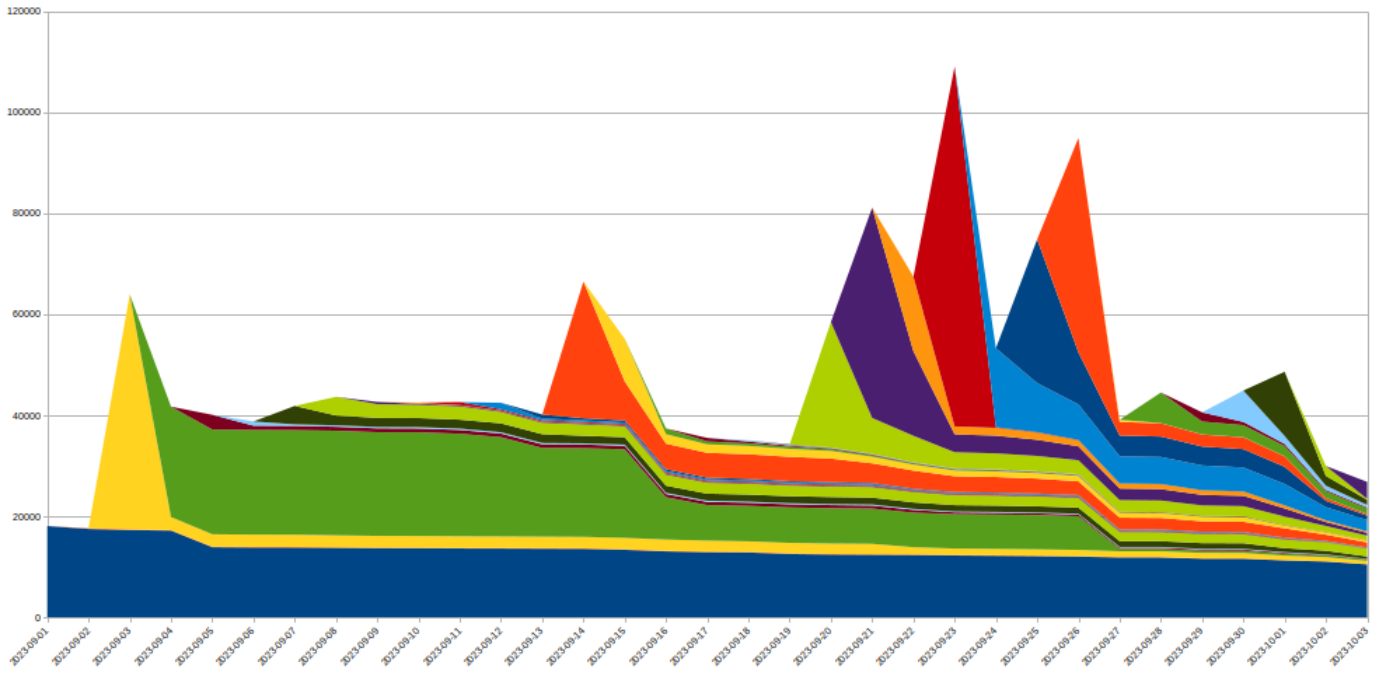
In addition to a direct fix, we have implemented several improvements to the server's HTTP/2 frame processing and request dispatch code. Furthermore, the business logic server has received improvements to queuing and scheduling that reduce unnecessary work and improve cancellation responsiveness. Together these lessen the impact of various potential abuse patterns as well as giving more room to the server to process requests before saturating.

Mitigate attacks earlier

Cloudflare already had systems in place to efficiently mitigate very large attacks with less expensive methods. One of them is named "IP Jail". For hyper volumetric attacks, this system collects the client IPs participating in the attack

and stops them from connecting to the attacked property, either at the IP level, or in our TLS proxy. This system however needs a few seconds to be fully effective; during these precious seconds, the origins are already protected but our infrastructure still needs to absorb all HTTP requests. As this new botnet has effectively no ramp-up period, we need to be able to neutralize attacks before they can become a problem.

To achieve this we expanded the IP Jail system to protect our entire infrastructure: once an IP is "jailed", not only it is blocked from connecting to the attacked property, we also forbid the corresponding IPs from using HTTP/2 to any other domain on Cloudflare for some time. As such protocol abuses are not possible using HTTP/1.x, this limits the attacker's ability to run large attacks, while any legitimate client sharing the same IP would only see a very small performance decrease during that time. IP based mitigations are a very blunt tool — this is why we have to be extremely careful when using them at that scale and seek to avoid false positives as much as possible. Moreover, the lifespan of a given IP in a botnet is usually short so any long term mitigation is likely to do more harm than good. The following graph shows the churn of IPs in the attacks we witnessed:



As we can see, many new IPs spotted on a given day disappear very quickly afterwards.

As all these actions happen in our TLS proxy at the beginning of our HTTPS pipeline, this saves considerable resources compared to our regular L7 mitigation system. This allowed us to weather these attacks much more smoothly and now the number of random 502 errors caused by these botnets is down to zero.

Observability improvements

Another front on which we are making change is observability. Returning errors to clients without being visible in customer analytics is unsatisfactory. Fortunately, a project has been underway to overhaul these systems since long before the recent attacks. It will eventually allow each service within our infrastructure to log its own data, instead of relying on our business logic proxy to consolidate and emit log data. This incident underscored the importance of this work, and we are redoubling our efforts.

We are also working on better connection-level logging, allowing us to spot such protocol abuses much more quickly to improve our DDoS mitigation capabilities.

Conclusion

While this was the latest record-breaking attack, we know it won't be the last. As attacks continue to become more sophisticated, Cloudflare works relentlessly to proactively identify new threats — deploying countermeasures to our global network so that our millions of customers are immediately and automatically protected.

Cloudflare has provided free, unmetered and unlimited DDoS protection to all of our customers since 2017. In addition, we offer a range of additional security