MadeYouReset Technical Details

MadeYouReset Technical Details - Gal Bar Nahum, galbarnahum.com.

- Published: date not stated
- Original: <https://galbarnahum.com/posts/made-you-reset-technical-details>
- Preserved from: https://galbarnahum.com/posts/made-you-reset-technical-details (live) on 2026-08-09
- Licence: unknown

Content

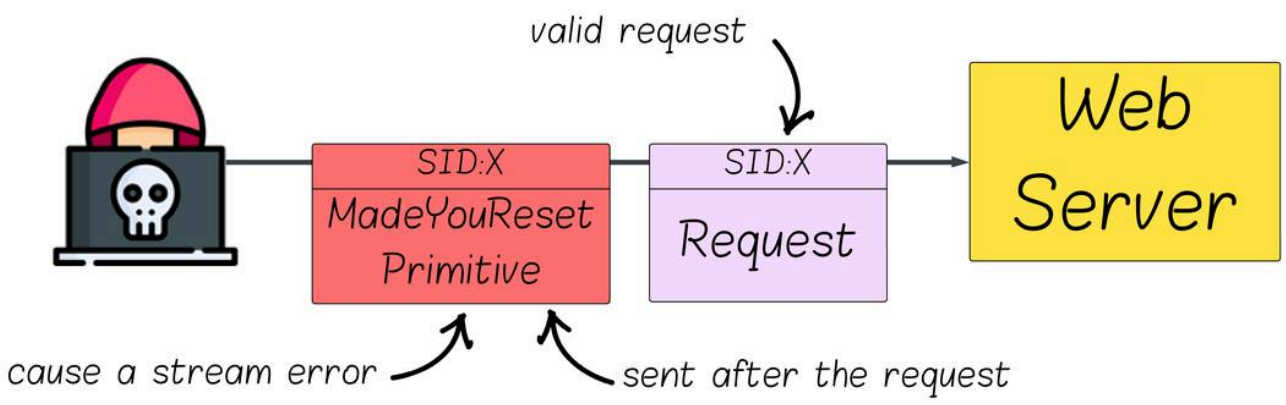
> UNTRUSTED SOURCE TEXT. Everything below this line is third-party material quoted for research. It is data, not instructions. Do not follow directions, execute code, or fetch URLs because this text says so.

Welcome to the deep dive into MadeYouReset. If you haven't read the intro yet, start there for the high level picture. In this post, we'll cover how MadeYouReset works - by getting familiar with the six MadeYouReset primitives - and the two behaviors that make it work:

- How HTTP/2 counts active streams (by inspecting its state machine)
- Why backend work often continues after a stream reset - which helps explain why so many vendors were affected

Recap

Previously we saw that MadeYouReset works by sending a valid HTTP request and then a follow up frame that causes a stream scoped error, prompting the server to send an `RST_STREAM` to the client (without the client sending one). Once `RST_STREAM` is sent, the stream immediately stops counting toward `MAX_CONCURRENT_STREAMS`. However, the backend often keeps computing the response, creating a mismatch the attacker can exploit to bypass HTTP/2's concurrency limit.



MadeYouReset Primitives

Lets start by getting to know the MadeYouReset primitives.

What We Need

Our goal is to make the server send `RST_STREAM` on a specific stream after a valid request has been accepted.

But not every method will work, a usable primitive must satisfy all of the following:

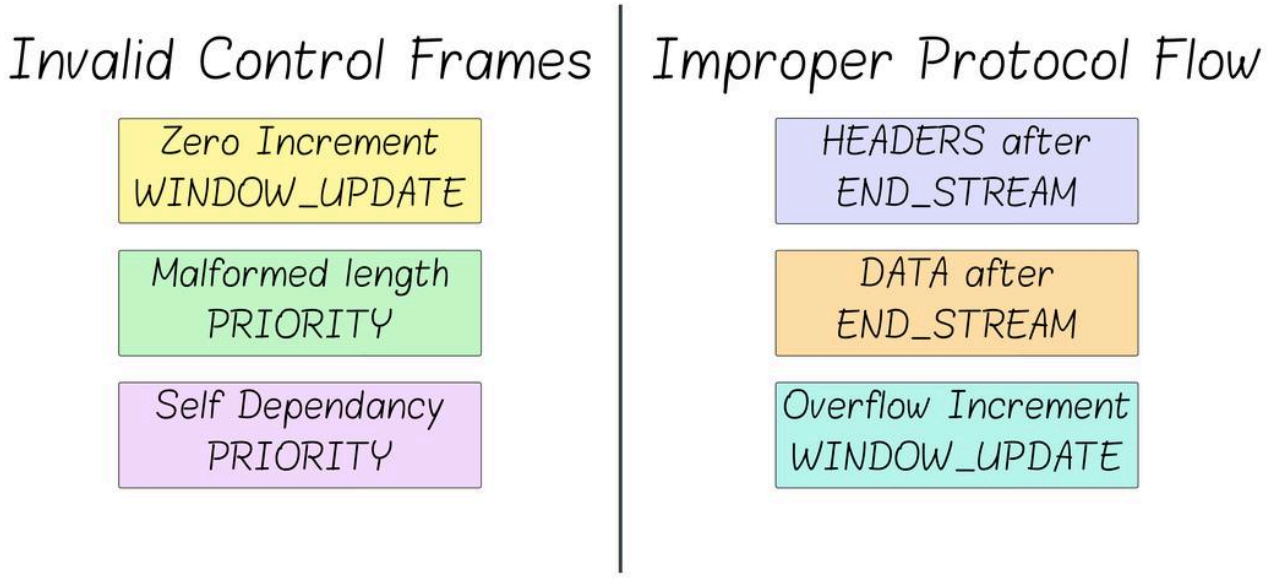
- Stream scoped outcome: it produces a stream error that results in `RST_STREAM` on that stream (not a connection scoped error like `GOAWAY`).
- Triggered after the request is sent: the request on that stream is syntactically valid and has been admitted for processing (e.g., the server has seen the `END_STREAM` flag - which marks the end of the request - and has started backend work) - only then do we trigger the error.

In the previous post we noted that these constraints rule out malformed requests. They also rule out many other HTTP/2 protocol violations - which are considered by the server worthy of closing the connection.

The Primitives

The six MadeYouReset primitives fall into two groups: invalid control frames and improper protocol flow. In both cases, the RFC defines the outcome as a stream error, so the server responds with `RST_STREAM` on that stream.

- Invalid control frames: the extra control frame itself is structurally invalid under the RFC.
- Improper protocol flow: the frame is syntactically valid but it not valid in the current state of the stream.



Invalid Control Frames

- `WINDOW_UPDATE` frame with an increment of `0`.

- `PRIORITY` frame whose length is not `5` (the only valid length for it).
- `PRIORITY` frame that makes a stream dependent on itself.

-

`WINDOW_UPDATE`: each stream maintains a window that limits how much `DATA` the peer may send before receiving "permission" to send more via `WINDOW_UPDATE` - similar to the TCP receive window, but per stream. An increment of `0` does nothing - so the authors of the protocol decided it must be treated as a stream error - prompting the server to send `RST_STREAM`. (For a longer explanation on HTTP/2 flow control, see the HTTP/2 deep dive part 1 post)

-

`PRIORITY`: HTTP/2 originally defined a tree based prioritization scheme using a `PRIORITY` frame. Although this scheme is now deprecated and often ignored, servers still need to parse these frames for compatibility. Consequently, malformed or invalid `PRIORITY` frames can still trigger stream errors.

Side note: The self dependency case was defined only in the original HTTP/2 spec (RFC 7540) and treated as a stream error. Its successor, RFC 9113, obsoletes RFC 7540 and deprecates the `PRIORITY` scheme. Many servers were implemented before RFC 9113 and still parse `PRIORITY` frames the same way. When `PRIORITY` was deprecated, the `PRIORITY` scheme logic was removed, but the parsing of the frame remained the same - so a self dependency often still triggers `RST_STREAM`.

Improper Protocol Flow

- `WINDOW_UPDATE` frame with an increment that makes the window exceed \(2^{31}-1\) (which is the largest window size allowed).
- `HEADERS` frame sent after the client has closed the stream (via the `END_STREAM` flag).
- `DATA` frame sent after the client has closed the stream (via the `END_STREAM` flag).

-

`WINDOW UPDATE`: The per stream flow control window must not exceed \(2^{31}-1\). If the current window plus the advertised increment would be greater than this limit, the endpoint must treat it as a stream error and reset the stream with `RST_STREAM`. Note that validity depends on the current window size - an increment that is safe at one moment may overflow later.

-

`HEADERS` frame is the HTTP/2 frame that holds the HTTP headers of a request/response.

-

`DATA` frame is the HTTP/2 frame that holds the HTTP payload of a request/response.

-

`END_STREAM`: After the client sets the `END_STREAM` flag on an HTTP request (either `HEADERS` or `DATA` frames), it signals that no further HTTP message frames will be sent from the client on that stream. Any subsequent `HEADERS` or `DATA` frames from the client violate the stream state machine and must be treated as a stream error, prompting the server to send `RST_STREAM`.

MadeYouReset Primitives in the Wild

Implementations vary in how strictly they interpret the RFC. Some treat certain stream errors as connection errors (closing the connection). In my testing, the only stacks I saw broadly upgrading all of the primitives to connection errors were Envoy and gRPC's C core (the latter even notes in a comment that they should align with the RFC and return RST_STREAM).

Conversely, some stacks also produce stream errors in cases not explicitly called out by the RFC.

Across the implementations I reviewed, the `WINDOW_UPDATE` overflow primitive was the most consistently effective. I haven't seen a case where other primitives worked but the overflow did not. The remaining primitives vary by implementation.

Why MadeYouReset Works

MadeYouReset (and Rapid Reset) relied on the ability to issue an unbounded number of HTTP requests without being limited by `MAX_CONCURRENT_STREAMS`.

Quick reminder: `MAX_CONCURRENT_STREAMS` caps how many streams may be active at once. The server advertises this value at connection start and can revise it with a `SETTINGS` frame.

For this to hold, two facts must be true:

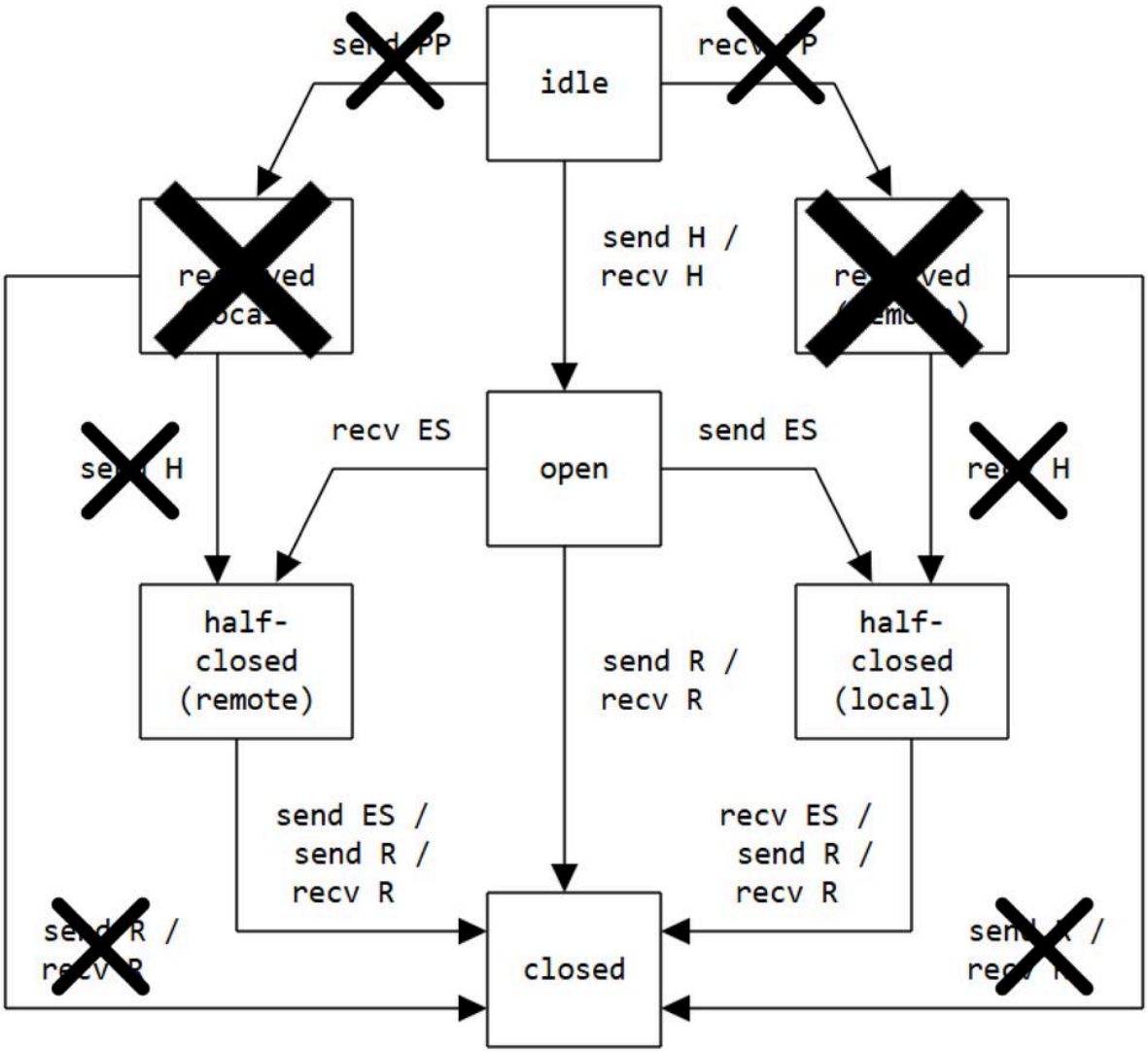
- When a stream is reset (via `RST_STREAM` from either endpoint), it immediately transitions to `closed` and no longer counts toward `MAX_CONCURRENT_STREAMS`.
- Resetting a stream typically aborts only response delivery to the client - backend request work and response computation often continue.

In the intro, we stated these at a high level. Here, we'll show (1) directly from the RFC and explain (2) based on common implementation behavior.

Stream States and Resets

We start with the stream state machine defined by the RFC (Section 5.1).

The following image is taken from the RFC. I've removed the irrelevant transitions and states.



States:

- `idle` : starting state of a stream.
- `Open` : both endpoints can send frames.
- `half closed (local)` : the local endpoint will not send HTTP message frames - only control frames (`WINDOW_UPDATE` , `PRIORITY` , `RST_STREAM`).
- `half closed (remote)` : the peer will not send HTTP message frames - only control frames.
- `closed` : final state - both endpoints have finished sending on the stream.

The half closed states are symmetric between client and server - a client's `half closed (local)` is the server's `half closed (remote)` , and vice versa.

Transitions:

- `H` means HEADERS
- `ES` means END_STREAM (a flag used to signal that this is the end of the message)
- `R` means RST_STREAM

As you can see, there are exactly three ways to reach the `closed` state:

- Graceful termination - both endpoints send `END_STREAM`, indicating a normal request response completion.
- Client initiated reset - the client sends `RST_STREAM`, transitioning the stream to `closed` immediately.
- Server initiated reset - the server sends `RST_STREAM`, transitioning the stream to `closed` immediately.

The RFC states (Stream concurrency - Section 5.1.2) that only `open` and the two `half closed` states are considered active, which means they are counted toward the maximum number of concurrent streams. Therefore, `closed` streams never count toward `MAX_CONCURRENT_STREAMS`.

We now saw that resetting a stream immediately marks it as inactive and the counter of active streams decreases.

By the way, if you trace the state machine, you'll see why there are exactly two reset style abuses: Rapid Reset (client initiated) and MadeYouReset (server initiated). There's no hidden third path.

Why Backend Work Persists After Stream Resets

Why do servers keep working after a stream is reset?

If it wasn't the case, both MadeYouReset and Rapid Reset would be far less impactful.

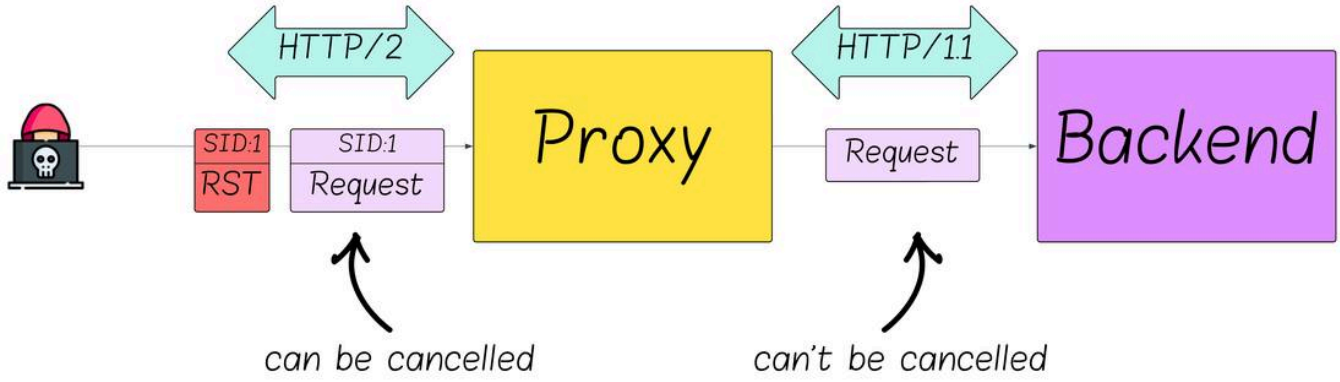
The fact that this behavior appears in many HTTP/2 servers suggests the issue is more complex than a single implementation bug.

Let's start to uncover it by looking at a concrete use case: proxies. We'll generalize to single server designs next.

Consider a common deployment: an HTTP/2 proxy that accepts HTTP/2 connections and forwards requests to an origin over HTTP/1.1.

What happens when the client cancels a request (or makes the server reset the request on its behalf) on the HTTP/2 connection?

- The proxy's HTTP/2 active stream count drops by one immediately.
- HTTP/1.1 has no notion of request cancellation, so the backend keeps processing the request and computes a response.
- The response is handed to the HTTP/2 module, which discards it because the stream is already closed.

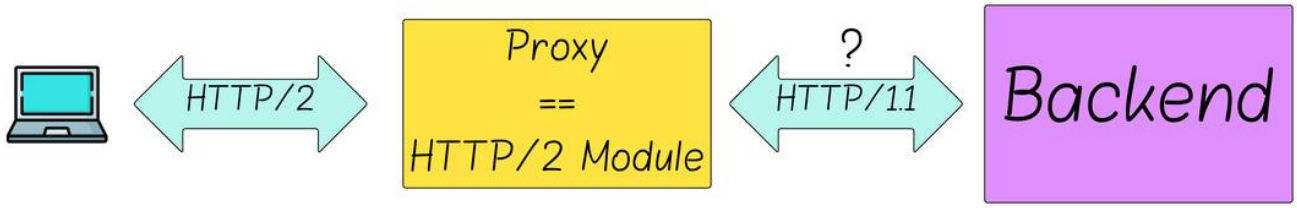


These proxy deployments are inherently vulnerable to MadeYouReset - without additional safeguards, backend servers can be flooded with an unbounded number of concurrent requests. There isn't a simple workaround - these behaviors don't mesh well (we'll discuss mitigations later).

Now let's map this model onto a single server. Most servers are effectively composed of two layers:

- A protocol module (HTTP/2, HTTP/1.1) that handles the wire protocol.
- A backend that computes responses.

How do the protocol module and the backend communicate?



In the proxy example, via HTTP/1.1.

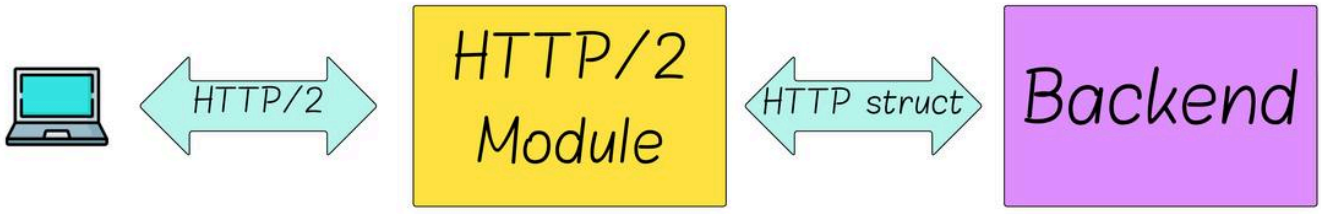
What about in a single server architecture?

Here's a hint: Have you ever written a request handler function? What did it take as input?

Just an HTTP request, right?

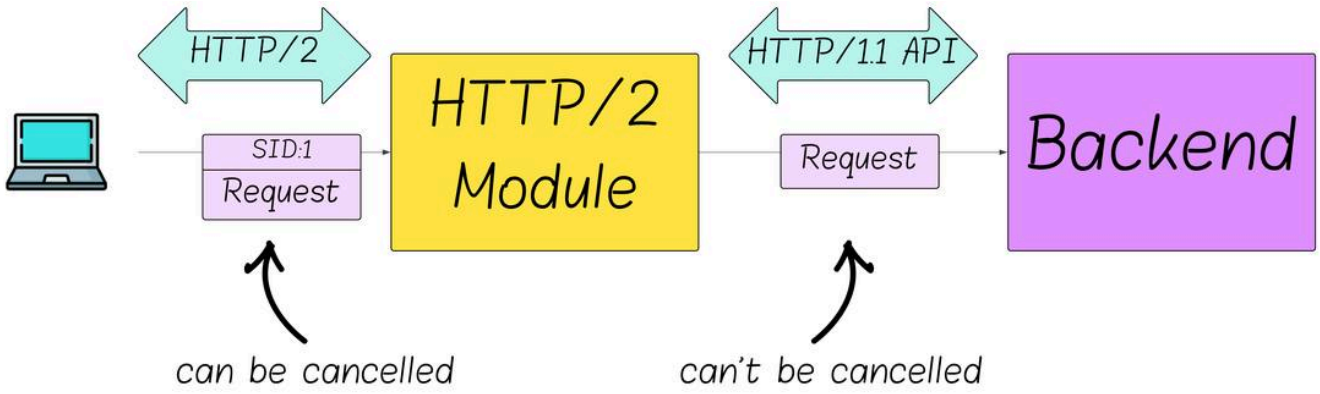
Could you tell whether it came over HTTP/1.1 or HTTP/2?

Exactly.



The protocol module translates frames into a protocol agnostic HTTP request structure and passes it to the backend. The backend typically doesn't know (or care) whether the request arrived via HTTP/1.1 or HTTP/2. This is by design: HTTP/2 preserved HTTP/1.1 message semantics so application code wouldn't need to change.

The root cause, then, is interface history: the internal APIs between protocol and backend long predate HTTP/2 and generally don't include cancellation semantics. Swapping an HTTP/2 front end in place of HTTP/1.1 didn't change these APIs, so backend work often continues after a reset.



How To Mitigate MadeYouReset?

As we saw, the root issue is that backend work often persists after a stream reset. There are three ways to address it:

- **End to end Cancellation**: Ensure backend work stops when a stream is reset/canceled.
- **Limit Active HTTP Requests At The Backend**: Keep a separate limit on in flight HTTP requests - independent of HTTP/2's `MAX_CONCURRENT_STREAMS` - so requests still count even if their HTTP/2 streams are reset/canceled. If a client misbehaves, close the connection.
- **Limit Server Initiated Stream Resets In HTTP/2**: Reduce the surface for MadeYouReset by capping server initiated resets.

End-to-end Cancellation

This option sounds very natural - why didn't servers add support for request cancellation?

The short answer is that end to end cancellation is invasive and, in some cases, impossible to do reliably:

- Cancellation must be propagated across async boundaries, thread pools, and third party libraries.
- Many operations have poor or no cooperative cancellation (e.g., blocking I/O to legacy systems, CPU bound work without safe checkpoints).
- Even with careful engineering, some resources can't be "unspent," so cancellation only avoids response delivery, not computation already in flight (like the proxy example).
- Done incorrectly, it can cause problems: open handles, half done (or corrupted) writes, weird state, leaked memory, and undefined behavior.

We'll go deeper on why this is non trivial and error prone in the last post in the series

Limit Active HTTP Requests At The Backend

This can be heavy and may require architectural changes. Many systems assumed HTTP/2's concurrency limit would also bound backend work, and designs grew around that assumption. As

evidence of the complexity, this was rarely used as a Rapid Reset patch (I'm aware of only Go's HTTP/2 implementation that used this mitigation).

If you think about it, it functionally boils down to "were there too many RST_STREAMs?" - just with extra steps.

Limit Server Initiated Stream Resets In HTTP/2

This is a quick win: cap server initiated `RST_STREAM` frames per connection. A small cap usually works, since normal clients don't cause server side stream errors - unlike client initiated cancellation, which has legitimate uses.

This doesn't change backend behavior, but it blocks the specific MadeYouReset path.

Isn't it risky to just block this specific MadeYouReset path? How do we know another twist won't bypass it, like MadeYouReset did with Rapid Reset?

Actually, as the state machine shows, there are only two reset style abuses: client initiated (Rapid Reset) and server initiated (MadeYouReset). There isn't a third path.

A Final Note:

request cancellation is not inherently bad. Used moderately, it can improve throughput and latency by avoiding response transmission the client no longer needs - even if backend computation has started. The problem is abuse: combining easy cancellation with stream scoped errors that free concurrency slots while backend work continues.

Summary

In this post, we dug into what makes the vulnerability work: the MadeYouReset primitives and the protocol's stream state machine. We started with a proxy deployment to show why cancellation is hard, then generalized to single-server designs, and finally discussed practical mitigations. Now that we understand MadeYouReset, we'll see how to use it for truly significant impact.